

# A Partitioning Methodology for BDD-based Verification

Debashis Sahoo<sup>\*</sup>, Subramanian Iyer<sup>†</sup>, Jawahar Jain, Christian Stangier,  
Amit Narayan, David L. Dill<sup>‡</sup>, E. Allen Emerson<sup>§</sup>

## ABSTRACT

The main challenge in BDD based verification is dealing with the memory explosion problem during reachability analysis. In this paper we advocate a methodology to handle this problem based on state space partitioning of functions as well as relations. We investigate the key questions of how to perform partitioning in reachability based verification and provide suitable algorithms. We also address the problem of instability of BDD based verification by automatically picking the best configuration from different short traces of the reachability computation. Our approach drastically decreases verification time, often in orders of magnitude.

## 1. INTRODUCTION

Verification and synthesis of sequential circuits require efficient techniques to represent and analyze the state space of the design under consideration [6, 10, 13]. It is well known that in sequential circuits the number of reachable states can be exponential in the number of state elements present in the circuit. A popular approach to deal with this *state explosion problem* consists of implicitly representing and manipulating functions using *Reduced Ordered Binary Decision Diagrams* (OBDDs) [2]. Though often efficient, there are cases, where the OBDD representation is not compact. Unfortunately, some practical application areas seem to exhibit this worst case complexity frequently. To overcome this problem of explosive memory requirements, the use of *Partitioned-OBDDs* (POBDDs) has been suggested [9]. By partitioning the state space into disjoint subspaces, and representing as well as processing all functions in each subspace independently of other subspaces, efficiency in time and space can be obtained.

The partitioned reachability techniques suggested in [11] do not sufficiently address the practical issues involved with partitioning, and as a result do not scale well on many difficult circuits. In [8], dynamically partitioned OBDDs were introduced as a capable data structure that extend the usefulness of POBDDs for reachability and model checking. In this paper, we address the various issues related to partition-

ing, including but not limited to data structure issues, and demonstrate techniques that perform better than OBDDs as well as classical Partitioned-OBDDs. These techniques use heuristics to improve the existing approaches.

Since OBDDs form a special case of POBDDs, where the whole function is represented in a single partition, we focus our attention on analyzing the deficiencies in the classical approach to partitioning. This will set the stage for introducing our algorithms and proving their effectiveness.

## What is missing in the classical approach?

Often OBDDs suffice for concise symbolic representation of boolean functions. Note, such functions are not the subject of this paper as further improving efficiency in their verification will not address the main bottleneck of the current BDD based verification. In such cases, an OBDD approach can be more efficient as they avoid the partitioning overhead. However, for many practical applications, the function representations are too large for efficient *monolithic* representation as a single OBDD. If we accept this premise, then partitioning should show a distinct advantage. In this context, some problems arise naturally, which have not been addressed effectively in the literature. For example, the key questions include what functions should be considered as a basis for generating partitions, how many partitions should be created, when should the partitioning commence, how should the processing of partitions be prioritized, etc. We posit that these questions are fundamental to creating any practical technique that exploits partitions, and hence, are fundamental for any technique to make the BDD based verification more practical and accordingly provide efficient algorithms to address the same.

Further, BDD approaches have a high sensitivity to parameter configuration. We develop a trace-centric approach to address this instability in BDD-based verification by automatically picking the best configuration from multiple short previews of the reachability computation.

Our fully automated approach completes all circuits but two in the VIS Verilog benchmark suite. An impressive gain over previous partitioned techniques is also seen.

## Related Work

Algorithms for POBDD-based reachability were presented in [11]. They do not however adequately address some of the key questions relating to the actual application of par-

<sup>\*</sup>sahoo@stanford.edu; Dept. Electrical Engineering, Stanford University, CA 94305, USA

<sup>†</sup>subbuk@cs.utexas.edu; Dept. Computer Sciences, University of Texas at Austin, TX 78712, USA

<sup>‡</sup>Stanford University

<sup>§</sup>The University of Texas at Austin; supported in part by NSF grants CCR-009-8141 and CCR-020-5483

tioning, which are addressed here.

In [4] a technique is discussed where the set of reachable states is decomposed into two or more sets during the intermediate stages of computation and reachability is performed on these decompositions separately. However, after a few steps of reachability, results from these different sets are combined to obtain a monolithic OBDD representation of the reachable state set.

Recently, a method for distributed model checking was studied by [7]. It parallelizes the classical model checking algorithm [5] using the window-based partitioning first mentioned in [11]. Such distributed techniques can further help increase the practicality of the approach presented here.

In the remainder of the paper we address the various questions raised above, present the results, and discuss their significance. We start by presenting relevant background information in Section 2. The basic rationale and an overall picture of our approach are in Section 3, followed by algorithmic details. Section 4 has experimental results that confirm that our approach is indeed more efficient and stable, in both time and space, than previous POBDD [11] as well as state-of-the-art OBDD approaches. Section 5 is a summary of the paper.

## 2. PRELIMINARIES

### Partitioned-OBDDs

The idea of partitioning was used to discuss a function representation scheme called Partitioned-OBDDs in [9], which was further extensively developed in [12].

1. [12] Given a Boolean function  $f : B^n \rightarrow B$ , defined over  $n$  inputs  $X_n = \{x_1, \dots, x_n\}$ , the partitioned-OBDD (henceforth, POBDD) representation  $\chi_f$  of  $f$  is a set of  $k$  function pairs,  $\chi_f = \{(w_1, f_1), \dots, (w_k, f_k)\}$  where,  $w_i : B^n \rightarrow B$  and  $f_i : B^n \rightarrow B$ , are also defined over  $X_n$  and satisfy the following conditions:

1.  $w_i$  and  $f_i$  are represented as OBDDs respecting the variable ordering  $\pi_i$ , for  $1 \leq i \leq k$ .
2.  $w_1 \vee w_2 \vee \dots \vee w_k = 1$
3.  $w_i \wedge w_j = 0$ , for  $i \neq j$
4.  $f_i = w_i \wedge f$ , for  $1 \leq i \leq k$

The set  $\{w_1, \dots, w_k\}$  is denoted by  $W$ . Each  $w_i$  is called a *window function* and represents a *partition* of the Boolean space over which  $f$  is defined. Each partition is represented separately as an OBDD and can have a different variable order. Most OBDD-based algorithms can be adapted easily for POBDDs.

Partitioned-OBDDs are canonical and various Boolean operations can be efficiently performed on them just like OBDDs. In addition, they can be exponentially more compact than OBDDs for certain classes of functions. The practical utility of this representation is also demonstrated by constructing OBDDs for the outputs of combinational circuits [12]. An excellent comparison of the computational power of various BDD based representations and partitioned-OBDDs may be found in [1].

## Reachability and Invariant Checking

The standard reachability algorithm is based on a breadth-first traversal of finite-state machines [6, 10, 16]. The algorithm takes as inputs the set of initial states,  $I(s)$ , expressed in terms of the present state variables,  $s$ , and a transition relation,  $T(s, s', i)$  that relates the next state  $s'$  a system can reach from a state  $s$  on an input  $i$ . The transition relation,  $T(s, s', i)$ , is obtained by taking a conjunction of the transition relations,  $s'_k = f_k(s, i)$ , of the individual state elements, i.e.,  $T(s, s', i) = \prod (s'_k = f_k(s, i))$ . Given a set of states,  $R(s)$ , that the system can reach, the set of next states,  $N(s')$ , is calculated using the equation  $N(s') = \exists_{s,i} [T(s, s', i) \wedge R(s)]$ . This calculation is also known as *image computation*. The set of reached states is computed by adding  $N(s)$  (obtained by replacing variables  $s'$  with  $s$ ) to  $R(s)$  and iteratively performing the above image computation step until a fixed point is reached.

State space partitioning induces a partitioning on transition relations. The transition relation,  $T_{jk}$ , comprised of transitions from states in partition  $j$  to states in partition  $k$ , can be derived by conjoining  $T$  with the respective window functions expressed appropriately in terms of present and next state variables, as  $T_{jk}(s, s', i) = w_j(s)w_k(s')T(s, s', i)$ . Each such  $T_{jk}$  can have an implicitly conjoined [3] representation.

```

POBDD-Reachability( $TR, InitStates$ ) {
  Initialize  $Rch$  to  $InitStates$ 
  Create partitioned rep for  $Rch$ 
  do {
    for (each partition  $i$ )
      Calculate LeastFixedPoint( $Rch$ ) in partition  $i$ 
    for (each partition  $i$ )
      Communicate states from  $i$  to all partitions
  } until (No new state is added to  $Rch$ );
}

```

Figure 1: POBDD-based Reachability Algorithm

The flow of the POBDD based-reachability algorithm is as shown in Fig. 1. Essentially, the algorithm performs as many steps as possible of image computation within each partition  $i$  using  $T_{ii}$ . This is called a step of *least fixed point* within the partition. When no more images can be thus computed, it synchronizes between partitions. This step is termed as *communication*, and is performed from partition  $i$  to each partition  $j$  using  $T_{ij}$ .

An invariant is a proposition that is to hold at every reachable state, and therefore invariants can be checked as newer states are added during the reachability computation.

In the next section, we will present techniques for the efficient construction of POBDDs. We address the issues of when, where, as well as how partitioning should be performed.

## 3. THE PARTITIONING METHODOLOGY

The problem of reachability is about representation of sets of states and relations, as well as operations performed on them. The key operation is successive image computation

on fragments of the state space until all reachable states have been explored. Thus, there is a need to develop an approach which can be efficient for both aspects – creating subspaces so as to represent functions succinctly as well as doing image computation.

In this context, the following questions naturally arise:

1. Is partitioning required at all?
2. If we must partition, what constitutes the “axis of partitioning”? In other words, along what lines should the partitioning be performed? For eg., what splitting variables should be used for creating windows?
3. As computation is performed, is the partitioning effective or is more partitioning required?
4. If the blowup is likely to be temporary (local), can the partitioning be likewise?
5. Once partitions are generated, in what order should they be processed?

These issues give more heuristic challenges on the POBDDs which can lead to a successful strategy in managing the behavior of BDDs in verification. Further due to the dynamic nature of partitioning, our approach can reduce the memory explosion in many circumstances. In contrast, the monolithic approach can exert no control on the program to prevent it from generating huge data structures that overflow memory.

We begin by discussing the algorithms for construction and utilization of partitioned representations, which address the questions raised above. Then, we detail the essential points of a trace-centric approach in the next section. This is used to impose some stability on the performance of the OBDDs with respect to the selection and setting of appropriate parameter values. At the end we give a complete reachability algorithm based on all the heuristics described in this paper. We now describe the mechanism for the construction and practical application of Partitioned OBDDs.

### 3.1 Whether to partition: Initial Partitioning

Since reachability needs manipulation of image BDD using transition relation, if either of them shows signs of blowup then partitioning seems to be the prudent choice. Figure 2

```

InitialPartitioning( $T, I$ ) {
  If ( $T$  is large) {
     $R := I$ 
    Do Partitioning using  $T$  as basis.
  } else {
     $R :=$  Do Reachability from  $I$  using  $T$  until Blowup.
    Do Partitioning using  $R$  and  $T$  as basis.
  }
  return Partitioned  $R$ ;
}

```

Figure 2: Initial Partitioning Algorithm

shows how partitioning is invoked. If the transition relation is small, then many initial steps of reachability identical to the classical approach using a single BDD can be performed and partitioning can be delayed. Reachability is performed using OBDDs until such time as a “blowup” in BDD-size is detected. This may be measured either absolutely as a maximum size of the symbolic representation of the image or in a relative way as the ratio of the representation of the reached states before and after any image computation. We adopt the latter approach with a threshold factor chosen *a priori*. However, if the transition relation cannot be easily constructed, then it is advantageous to partition quickly.

### 3.2 How to Partition: Choice of partitioning variable

After a “blowup” is detected, we select  $n$  splitting variables and the corresponding  $2^n$  partitioning windows are created. The choice of the splitting variables is critical to the effectiveness of the partitioning approach. The goal is to create small and relatively balanced partitions that represent *non-compatible* functions. A set of functions is said to be *non-compatible* if the totality of their individual representations using different orders is far more compact, than their combined representation as a whole. The splitting variable is selected by means of a cost function, for e.g., as described in [11]. For each variable, the cost function takes into account the relative BDD-sizes of the positive and negative co-factors with respect to the BDD-size of the original graph.

```

SelectPartitioningVars(basis BDD  $F$ ) {
  for (each method  $i := 1$  to  $m$ ) {
    get ordered splitting variable list using  $F$ .
    select top  $k$  variables.
    for (each subspace  $j := 1$  to  $2^k$ ) {
       $cost[i][j] :=$  size of the cofactor  $F_j$ .
    }
    cost of method  $i := \sum_j cost[i][j]$ 
  }
  select method with lowest cost.
  return corresponding vars.
}

```

Figure 3: Selecting Partitioning Variable

However, the measurement of graph sizes for determining a blowup and for recognizing its subsidence can be done with respect to the BDD-size of the transition relation or the image representation or both. We try to get separate splitting variable choices from each of these three methods. We select that choice which gives the smallest co-factor graphs after reordering as illustrated in the Figure 3. Intuitively, this selects a variable that creates two partitions as non-compatible as possible.

### 3.3 Are more partitions required: Global Dynamic Repartitioning

Whenever a BDD-size blowup is detected during computation in a partition, dynamic repartitioning [8] is performed, as illustrated in Figure 4. Repartitioning is performed by splitting the given partition by co-factoring the entire state

space based on one or more suitable, newly calculated, splitting choices until the blowup has been ameliorated. Initially, the partitioning is done using one splitting variable. To prevent excessive overhead in the new splitting variable selection, they are obtained by recalculating the cost of only the top few choices provided by the partitioning variable selection method discussed before. At this point, each new partition is checked to see whether the blowup has subsided. If not, repartitioning is recursively performed on that partition. A threshold on maximum number partitions is kept to prohibit the method to produce exponential number of partitions.

```

DynamicPartition(basis BDD  $F$ , partition  $i$ ){
   $v := \text{SelectPartitioningVars}(F)$ 
  create partition  $i_1$  from  $i$  with  $v := 0$ 
  if (blowup in  $i_1$ )
    DynamicPartition( $F_{v:=0}$ ,  $i_1$ )
  create partition  $i_2$  from  $i$  with  $v := 1$ 
  if (blowup in  $i_2$ )
    DynamicPartition( $F_{v:=1}$ ,  $i_2$ )
}

```

Figure 4: Dynamic Partitioning

It must be noted that the variable selection algorithm ensures that superfluous partitions are not created and that the ones created are somewhat balanced. In practice this imposes a bound on how many partitions are actually created.

```

ComputeImage( $TR$ , state set  $R$ , variable list  $L$ ){
  do {
    one microstep of image
    if (blowup) {
       $varList := \text{top } k \text{ vars from } L$ 
      create partitions using  $varList$ 
      for (each new partition)
        recursively do all remaining micro steps
    }
  }while(microsteps remain)
}

```

Figure 5: Computing Image with Local partitioning

### 3.4 Partition only Image: Local Partitioning

During each step of image computation, many steps of alternating composition and conjunction are performed. Often it is found that the blowup in the BDD sizes during such a *micro-step* of image computation is a temporary phenomenon which eventually subsides by the time the image computation is completed. In such a case the invocation of dynamic global repartitioning could create a large number of partitions, whose BDD-sizes become eventually very small. Hence, it is advantageous to create these partitions *locally* only for that particular image computation and then recombine them before the end of the image computation. If local partitioning does not reduce the blowup, then dynamic global repartitioning can be done. To create the local partitions, we cofactor using the ordered list of splitting variables

that was generated earlier. Figure 5 describes how this is done.

### 3.5 How to order partitions: Scheduling

In this section we describe our technique for state space traversal which schedules partitions based on their difficulty of traversal. The expectation is that the probability of catching an error is higher as more of the state space is covered. We characterize partitions in terms of how quickly it has been possible to cover state space symbolically in that partition. This is measured in terms of a cost for processing the partitions. The details of how this cost is computed is described in the following. Once this characterization of the level of difficulty is available, we schedule the partitions for processing in ascending order of their costs. Thus, the state space can be explored in a way that speeds up the rate at which new states are discovered. Notice that in the “worst” case, this processes all the partitions and thus traverses the entire state space if the design is correct

```

Reachability( $T, I$ ) {
   $R := \text{InitialPartitioning}(T, I)$ 
  Initialize Priority Queues in Scheduler  $S$ ;
  do {
    Get LFPList from  $S$ .LFPQueue
    for each partition  $i$  in LFPList
      Calculate LeastFixedPoint in  $i$  and update  $S$ 
    Get CommList from  $S$ .CommQueue
    for each partition  $i$  in CommList
      Communicate from  $i$  to all parts and update  $S$ 
  }until (No new state is added to  $R$ );
}

```

Figure 6: Scheduling-based Reachability Algorithm

#### 3.5.1 Scheduling Cost metrics

We will now describe two metrics that are used for assigning a scheduling cost for processing the partitions.

**Density based scheduling:** Similar to [14] we define the *density* of a partition as the ratio of the number of reachable states discovered in that partition to the size of the BDD representing the reachable states. It may be noted that large function representation sizes, i.e. BDD-sizes, are the most important bottleneck in symbolic verification techniques. Thus, in the interest of greater and faster state space coverage, it is advisable to first process partitions with a higher *density*.

**Time based scheduling:** Note that each partition may require many fixed point computations. Hence, another useful metric takes into account the time required for the latest fixed point computation within each partition. The partition with faster fixed point computation is intuitively more attractive as it may be more amenable to symbolic manipulation using BDDs. Therefore, it is advantageous to select partitions which have historically been known to take lesser time. In the above calculations the time spent in communicating either to or from any partition was excluded.

The cost for processing a partition is the ratio of the time taken for the most recent fixed point computation to the

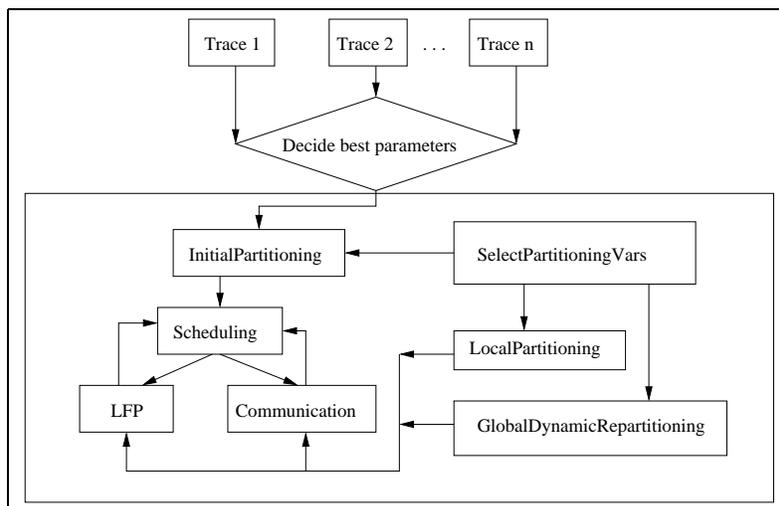


Figure 7: Trace-based Reachability Algorithm

density of that partition. Intuitively, this prioritizes partitions that are more amenable to symbolic traversal. In our reachability algorithm (Figure 6), priority queues are used to schedule partitions in increasing order of their cost.

#### 4. ADDRESSING INSTABILITY IN BDD-BASED VERIFICATION

Instability in BDD based verification refers to the sensitivity of performance to various parameters like the size of the clusters in the implicitly conjoined transition relation, the selection of variable reordering methods etc. It is observed that a single choice seldom works uniformly for all cases and therefore, such parameters need to be tweaked manually. The performance of BDD based methods can vary widely and unexpectedly based on these settings.

In a partitioned scheme, there are an even greater number of specific choices available for state space traversal. The degrees of freedom include the number of partitions, when and whether to dynamically decompose partitions further, how to schedule the image computations involving multiple partitions, etc. Hence, we propose a *trace-centric* approach to parameter selection which can dynamically fine-tune the partitioning choices and balance the various options available to a BDD based method.

A small set of identical computations are separately executed, each using a different choice of the various options. Each of these is referred to as a *trace*. The length of a trace is how far it proceeded into the entire computation. A large number of traces may lead to a high cost in overhead. Therefore, we elect to look at just a few traces with orthogonal settings. These traces are only observed until the size of the OBDDs exceeds a pre-determined threshold.

The traces are compared with one another on various factors, for e.g., the blowup of OBDDs when performing the image operation, the number of image operations completed, number of states traversed, etc. The configuration for the full run is adopted from that of the most efficient trace. Needless to say that if a trace completes the reachability

in the allowed space and time, no further computation is required. We have found that even very simple dynamic examinations can be dramatically effective in stabilizing the performance of BDDs.

Also, we find that the overhead of generating multiple traces is minor when balanced against the savings. Even if graph size is reduced by a factor of 2 (in the more efficient configuration), it proves to be important. This is because during the reachability multiple reorderings are triggered, and even saving one large reordering of 1 million node graph compensates calculation of 3 different traces with maximum BDD threshold of 100k BDD nodes.

#### 5. THE COMPLETE REACHABILITY ALGORITHM

The complete reachability algorithm is shown in Figure 7. The input to the reachability algorithm is a transition relation  $T$  and initial states  $I$ . The algorithm first picks a best parameter configuration by running few short traces with orthogonal settings. Then it runs the *InitialPartitioning* procedure described in section 3.1. After this the algorithm performs POBDD-based state traversal guided by the scheduling heuristics described in section 3.5. There are two important steps in the POBDD based state traversal algorithm. The first one is computing a fixed point (called LFP) inside a partition and the other one is to compute image of a function in other partitions (called Communication from one partition to other partitions). The scheduler selects the partition to process next for the above operations. The scheduler implements two priority queues, one for each of the above operations described. Each partition is assigned a cost described in section 3.5. The local partitioning described in section 3.4 and the global dynamic repartitioning described in section 3.3 is enabled during the fixed point and communication. The *SelectPartitioningVars* heuristics described in section 3.2 is called when partitioning is needed.

## 6. EXPERIMENTS

Our implementation of the POBDD-data structure and algorithms uses VIS-2.0, which is a state-of-the-art public domain BDD-based formal verification package. We have chosen VIS for its Verilog support and its powerful OBDD-package (i.e. CUDD [15]). As our techniques affect only the BDD-data structures and algorithms, they can – with moderate effort – be implemented in other packages as well. These techniques work with any method of image computation; for this implementation, both OBDDs and POBDDs use the IWLS95 method.

We found that `lazy_sift` reordering method works better for most cases. All the experiments use `lazy_sift` BDD reordering method.

### Benchmarks

For experiments on reachability and invariant checking, we chose various public domain circuits: the VIS-Verilog [17] benchmark suite and ISCAS89 benchmark suite. We choose only invariant checking properties in VIS-Verilog benchmark suite. For sake of brevity, results are omitted for the smaller examples and presented only on those circuits where VIS requires more than 250,000 BDD nodes.

### Results

We compare the methodology proposed in this paper with three other approaches: the non-partitioned approach of VIS, the static partitioning approach and our own partitioning approach without computation traces. We find that the computation of small traces outperforms, sometimes significantly, all other approaches.

### Vs. Non-partitioned approach, Invariant Checking

Table 1 compares the non-partitioned approach of VIS with the proposed method on the time and space needed to check invariant properties from the VIS-Verilog benchmark suite. The time includes cpu time for simultaneous check of all properties of a given circuit. The memory required is measured in terms of the cumulative peak live nodes for all BDDs that are maintained.

The first column shows the memory required when running VIS. The second column lists the memory for our trace-centric partitioning method, and the next column shows the corresponding space gain. In the runtime comparison, the first column shows time taken by VIS in seconds. The second column lists the effect of our improved partitioning method when combined with a trace-centric approach. The last column shows the time gain of the trace centric partitioning over VIS.

In both time as well as space, the trace-centric partitioning approach provides dramatic gains. Our approach completes all circuits except two in the VIS Verilog benchmark suite. Notably, it verified six circuits where the VIS failed to finish. For some circuits such as *palu*, *s1269b*, *am2910*, *ball*, verification was completed by the very first POBDD trace of 100k nodes. In most cases, there is an order of magnitude

or more improvement in both time as well as in space.

### Vs. Non-partitioned approach, Reachability Analysis

In identical format, Table 2 compares our method with Vis-2.0 on formal reachability analysis for some ISCAS89 benchmark circuits. The proposed POBDD implementation works

	Space (BDD nodes)			Time (sec)		
	Vis	Proposed	Gain	Vis	Proposed	Gain
s1269	2.4M	31K	77	2305	28	82
s3330	1.3M	263K	4.9	T/O	948	>92
prolog	976K	138K	7.1	T/O	592	>147
s4863	438K	264K	1.7	1382	1717	0.8
s1423	3.3M	1.7M	1.9	T/O	T/O	-
	States covered			2e+10	1e+13	419
	Time for 2.3e+10 states			87000	633	137

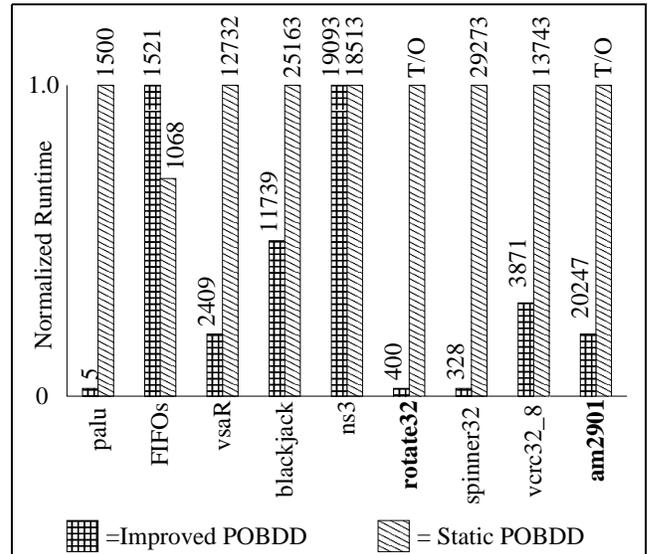
**Table 2: Comparison of OBDD-VIS with POBDD-VIS on ISCAS89 benchmark**

better in first three circuits. In s4863, the time required in computing traces made the method slightly slower than VIS.

We also found that in s1423 our method covers more states than VIS does in the same time. Also notice that the partitioned approach covers the same number of states as VIS in a small fraction of the time required.

### Vs. Static Partitioning

Figure 8 compares the run time of our trace-centric POBDD method to the static POBDD approach of [11]. The initial



**Figure 8: Comparison of Dynamic trace-centric and Static Partitioning Approaches on all Large designs (time>1000s) of VIS-benchmarks. The actual runtime in seconds is shown at the top of each bar.**

number of partitions for both methods were kept identical to

	Space (BDD nodes)			Time (sec)		
	Vis	Proposed	Gain	Vis	Proposed	Gain
palu	371K	7K	53.0	186	5	37.2
s1269b	2.6M	38K	68.4	1189	27	44.0
sp_product	919K	70K	13.1	1299	440	3.0
FIFOs	975K	131K	7.4	1704	1521	1.1
vsaR	5.2M	1.3M	4.0	5281	2409	2.2
blackjack	3.2M	1.1M	2.9	16298	11739	1.4
ns3	4.7M	1.0M	4.7	18592	19093	1.0
am2910	11.7M	67K	>174	M/O	222	>392
ball	18.8M	17K	>1106	T/O	168	>518
spinner32	1.4M	248K	> 5.6	T/O	335	>260
rotate32	827K	240K	> 3.4	T/O	293	>297
vcrc32_8	20.5M	2.4M	> 8.5	T/O	3871	>23
am2901	20.8M	2.9M	> 7.2	T/O	20247	> 4.3
b12	4.2M	800K	> 5.3	T/O	T/O	–
vsa16a	11.1M	4.8M	> 2.3	T/O	T/O	–

**Table 1: Comparison of OBDD-VIS with POBDD-VIS on VIS-benchmarks for all circuits where VIS requires more than 250K BDD nodes (T/O = timeout = 1 day, M/O = memout = 512MB). The time includes cpu time for simultaneous check of all properties of a given circuit**

have a level playing field. The graph shows the normalized runtimes by size of the bar. The actual runtime in seconds is shown at the top of each bar. One can observe that the proposed method noticeably improves on the static partitioning scheme for most of the circuits, especially when the time taken is large. In once case that could not be completed by the static partitioning approach, the current method is able to complete reachability.

### Traces Vs. No traces

Table 3 compares proposed POBDD approach with and without traces. The proposed POBDD with trace finishes one

	Time (sec)		
	Vis	Proposed (without trace)	Proposed (with trace)
am2910	M/O	222	222
ball	T/O	168	168
spinner32	T/O	9305	335
rotate32	T/O	5537	293
vcrc32_8	T/O	51576	3871
am2901	T/O	T/O	20247
b12	T/O	T/O	T/O
vsa16a	T/O	T/O	T/O

**Table 3: Comparison of Proposed POBDD with trace and without trace on all designs where VIS runs out of time or memory.**

more circuit that the method without trace. It has notable improvements on three other circuits, viz., *spinner32*, *rotate32*, *vcrc32\_8*. Table 3 shows that the partitioning methodology is definitely improved by using short traces.

## 7. CONCLUSIONS

We have discussed an efficient methodology for improving difficult instances of reachability based verification using the approach of state space partitioning. We have investigated relevant problems posed in creating a partitioned data structure during BDD-based verification, and provided efficient and practical algorithms for the same.

We have also addressed the issue of instability in BDD based approaches where parameters are seldom found to work well uniformly. We developed a trace-centric approach to selection of such parameters. The resulting method dramatically improves the space and run time, often from one to three orders of magnitude, on various public-domain benchmark circuits that are otherwise known to be difficult.

It is found that methods based on a monolithic representation of the state sets often encountered space explosion early on in the computation, after which they could not make much progress due to memory limitations. However, the trace-centric partitioning method scaled well, and could finish most circuits in the VIS Verilog benchmark suite.

## 8. REFERENCES

- [1] B. Bollig and I. Wegener. Partitioned BDDs vs. other BDD models. In *IWLS*, 1997.
- [2] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.
- [3] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic Model Checking with Partitioned Transition Relations. In *Proc. of the Design Automation Conf.*, pages 403–407, 1991.
- [4] G. Cabodi, P. Camurati, and Stefano Quer. Improved reachability analysis of large finite state machines. *ICCAD*, pages 354–360, 1996.
- [5] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time

- temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.
- [6] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines based on symbolic execution. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, 1989.
- [7] Orna Grumberg, Tamir Heyman, and Assaf Schuster. Distributed symbolic model checking for  $\mu$ -calculus. In *Computer Aided Verification*, pages 350–362, 2001.
- [8] S. Iyer, D. Sahoo, C. Stangier, A. Narayan, and J. Jain. Improved symbolic Verification Using Partitioning Techniques. In *Proc. of CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, 2003.
- [9] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham. Functional partitioning for verification and related problems. *Brown/MIT VLSI Conference*, 1992.
- [10] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [11] A. Narayan, A. Isles, J. Jain, R. Brayton, and A. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *ICCAD*, pages 388–393, 1997.
- [12] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned-ROBDDs - A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions. In *ICCAD*, pages 547–554, 1996.
- [13] C Pixley. A theory and implementation of sequential hardware equivalence. *ICCAD*, 11, 1992.
- [14] K. Ravi and F. Somenzi. High-density reachability analysis. In *ICCAD*, pages 154–158, 1995.
- [15] Fabio Somenzi. CUDD: CU Decision Diagram Package <ftp://vlsi.colorado.edu/pub>, 2001.
- [16] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *ICCAD*, pages 130–133, 1990.
- [17] VIS. Vis verilog benchmarks <http://vlsi.colorado.edu/vis/>, 2001.