Decomposed Symbolic Forward Traversals of Large Finite State Machines

Stefano Quer † Gianpiero Cabodi † Paolo Camurati ‡

† Politecnico di Torino
Dip. di Automatica e Informatica
Turin, ITALY

‡ Università di Udine
Dip. di Matematica e Informatica
Udine, ITALY

Abstract

BDD–based symbolic traversals are the state-of-the-art technique for reachability analysis of Finite State Machines. They are currently limited to medium–small circuits for two reasons: BDD peak size during image computation and BDD explosion for state space representation. Starting from these limits, this paper presents a technique that decomposes the search space decreasing the BDD peak size and the number of page faults during image computation. Results of intermediate computations and large BDDs are efficiently stored in the secondary memory. A decomposed traversal that allows exact explorations of state spaces is obtained. Experimental results show that this approach is particularly effective on the larger MCNC, ISCAS'89, and ISCAS'89–addendum circuits.

1 Introduction

Finite State Machines (FSMs) are a popular model for *control–dominated* ASICs. FSMs are identified by their input/output alphabets, initial state sets, and next state and output functions. Exploring the state space of a FSM allows us to prove many useful properties. Among them, let us list *equivalence*, *resetability*, and *synchronization*.

A *forward traversal* of a FSM identifies its *reachable state space*. Intuitively, a state is reachable if a sequence of inputs causes the FSM to evolve from any initial state to that state. The next state function determines the evolution along time. The next states are the image, for all inputs, of the current state set according to the next state function. Next states are repeatedly added to previously reached states. The process terminates as soon as either a fixed-point is reached, i.e., no newly reached states are found, or the property under scrutiny doesn't hold.

Symbolic approaches compute images implicitly enumerating the inputs [1], [2] using the "*transition relation*" or the "*transition function*".

The definition of image is straightforward if a "*transition relation*" [1] describes sequential behaviour. The relation is the set containing all the couples (current state s – next state y) such that there is at least an input value x that lets the FSM evolve from state s to state y. Computing the image of a set of states comes down to considering only those pairs whose current state belongs to the set, returning the corresponding next state. The operations are conjunctions of BDDs and existential quantification of the current states.

Although quite successful, symbolic methods cannot complete the reachability analysis of large FSMs, because they require too much memory and are computationally too expensive. Focussing on memory requirements, the two major problems are the BDD peak size during image computation and the size of the BDD representing reached states. Virtual memory should be considered a good solution to such a problem but, if the working set size for a program is large and the memory access are random, an enormous number of page faults significantly modify the performance of the program.

Our approach consists in decomposing sets of states and in carrying out forward traversal in a decomposed form. When monolithic BDDs become too large or when image computation becomes too expensive, we decompose sets in subsets. This allows us to deal with just one subset at a time, decreasing both the number of BDD nodes and the peak size. Fixed point computation and other operations can be carried out on the decomposed form.

Good decomposition of state sets is a key issue. A simple and fast set decomposition technique, that has almost no overhead in terms of nodes and of CPU time is proposed.

Different state sets have in general mutually overlapping images and this entails a re-computation of the states that belong to more than one image. Despite this possibility, as the cost of image computation depends more than linearly on the size of the current state set, decomposition decreases the overall complexity, with remarkable experimental benefits. This is especially true for large circuits but it is also evident on smaller ones.

The decomposed approach has other advantages over the standard one:

- it exploits mass memory to download large BDDs and the results of intermediate computations
- it resorts to compression techniques to store BDDs
- it allows different dynamic ordering strategies to be applied to each decomposition.

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00 © 1996 IEEE

Coudert and Madre [5] resort to a full domain or co-domain partitioning for image computation. They applied these techniques to the transition function and recursively decomposed the problem until a terminal case was found.

Ravi *et. al.* [4] use a mixed breadth–first/depth–first traversal focused on *dense* BDDs applied to the transition function method. The objective of the strategy is to take large *subsets* of the state sets representable with a small number of BDD nodes. This technique has been quite effective in finding large portions of the reachable state space, but it overestimates the distance of a state from the reset states and is unable to identify the sequential levels of the circuit.

Our approach is based on the transition relation and partitions BDDs until their size is lower than a given threshold, not until a terminal case is reached. The method is *exact*, as it computes the set of reachable states, not an over– or under–estimation, and works at its best when the reachable state set is represented by a "*uniform*" BDD. Experimental results show that it is particularly effective on the larger ISCAS'89 and ISCAS'89–addendum circuits.

The remainder of the paper is organized as follows. Section 2 summarizes some useful concepts. Section 3 describes enhanced symbolic traversal based on decomposition and section 4 describes the decomposition procedure. Section 5 shows some experimental results. Section 6 closes the paper with a brief summary and reports on future work.

2 Preliminaries

2.1 The Model

A *finite state machine* is an abstract model describing the behavior of a sequential circuit. A completely specified FSM M is a 6-tuple $M = (I, O, S, \delta, \lambda, S_0)$, where $\mathcal{B} = \{0, 1\}$, $I = \mathcal{B}^m$ is the input alphabet, $O = \mathcal{B}^o$ is the output alphabet, $S = \mathcal{B}^n$ is the state space, δ is the next state function ($\delta : S \times I \rightarrow S$), λ is the output function ($\lambda : S \times I \rightarrow O$), and S_0 is the initial state set. Let $x = (x_1, x_2, \ldots, x_m)$, $s = (s_1, s_2, \ldots, s_n)$, and $y = (y_1, y_2, \ldots, y_n)$ be vectors of Boolean variables describing the input variables I, the state variables S and the output variables O, respectively.

2.2 Sets and Characteristic Functions

Let A be a subset of $\mathcal{B}^n$. The *characteristic function* of A is the function $\chi_A : \mathcal{B}^n \rightarrow \mathcal{B}$ defined by:

$$\chi_A(a) = \begin{cases} 1 & \text{if } a \in A \\ 0 & \text{if } a \notin A \end{cases}$$

Set operations are efficiently implemented by Boolean operators on BDDs. With abuse of notation, in the rest of this paper we make no distinction between the BDD representing a set of states, the characteristic function of the set, and the set itself.

2.3 The Transition Relation

Let us describe the sequential behaviour of a FSM as a relation [1], defining a characteristic function $\chi_\delta(x, s, y) : \mathcal{B}^m \times \mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}$ that returns 1 iff the next state $y \in S$ is the image of the current state $s \in S$ and of the input $x \in I$ according to δ. As we are concerned by the existence of an input, rather than by its value, the *transition relation* abstracts from the inputs. Let M be a FSM. The *transition relation* T_M associated to M is:

$$T_M(s, y) = \exists_x \prod_{i=1}^{n} (y_i \equiv \delta_i(x, s)) = \exists_x \prod_{i=1}^{n} t_i(x, s, y)$$

2.4 Image of a Set

Let $f : \mathcal{B}^i \rightarrow \mathcal{B}^j$ be a Boolean function and $C \subseteq \mathcal{B}^i$ a subset of its domain. The *image* of C according to f is:

$$\textsc{Img}(f, C) = \{y \in \mathcal{B}^j \text{ such that } \exists x \in C \ \wedge \ y = f(x)\}$$

Subset C is often called "*constraint*". Whenever $C = \mathcal{B}^i$, the image is often called "*range*".

In particular the image of a set of states described by its characteristic function $C(s)$ according to δ using the transition relation is defined as:

$$\textsc{Img}(\delta, C(s)) \quad = \quad \exists_s \ (T_M(s, y) \cdot C(s))$$

2.5 Symbolic Traversal

A *Symbolic Traversal* (fig. 1) is a breadth–first search that returns at each iteration the set of states T reached from the current one F. This is accomplished by means of a symbolic image computation $\textsc{Img}(\delta, \mathsf{F})$. Set N contains the T states that have not yet been visited. Reached states are accumulated in R. The starting set F is initially set to S_0 and then is selected choosing a suitable BDD that represents all newly reached states and possibly some of the already visited ones, as in [2] (procedure BEST_BDD).

The termination condition is to find a least fixed-point. This condition is equivalent to testing the emptiness of N at each step.

Explicit enumeration has complexity equal to the total number of states in the FSM whereas the complexity of the symbolic algorithm is lower. Large amount of states, greater than 10^{120}, have been visited efficiently with this method.

3 Decomposed Symbolic Traversal

Proceeding from one step to the next one in symbolic traversal, all the sets, in particular F and R, become larger and much more complex as new variables are added to their supports.

Symbolic traversal experiences two bottlenecks:

- a monolithic BDD representing the set may be too large
- it may be impossible to perform an image computation (function IMG), because of the size of the BDDs involved in intermediate computations.

Our approach consists in decomposing state sets when, during traversal, they become too large to be represented as a monolithic BDD or when image computation becomes too expensive.

The basic idea is that, if we decompose the current state set $C(s)$ as $C_1(s) + C_2(s)$, its image according to δ is equivalent to the union of the images of its decompositions $C_1(s)$ and $C_2(s)$. In fact working with the transition relation we can write:

$$\begin{aligned}
\textsc{Img}(\delta, C(s)) &= \exists_s (T_M(s,y) \cdot C(s)) \\
&= \exists_s \exists_x (\textstyle\prod_{i=1}^{n} t_i(x,s,y) \cdot C(s)) \\
&= \exists_s \exists_x (\textstyle\prod_{i=1}^{n} t_i(x,s,y) \cdot (C_1(s) + C_2(s))) \\
&= \exists_s \exists_x (\textstyle\prod_{i=1}^{n} t_i(x,s,y) \cdot C_1(s)) \\
&\quad + \exists_s \exists_x (\textstyle\prod_{i=1}^{n} t_i(x,s,y) \cdot C_2(s)) \\
&= \textsc{Img}(\delta, C_1(s)) + \textsc{Img}(\delta, C_2(s))
\end{aligned}$$

After a decomposed image computation, we can recompose the resulting set or we can carry on traversal in a decomposed form.

Sets can be split not only at a traversal step but also inside image computation, optimizing "atomic" conjunction–abstraction operations.

Figure 2 shows the pseudo-code for the decomposed traversal. Initially, R_{dec} and F_{dec} are set to S_0. Parameter *n_dec* indicates the current number of decompositions and is initially set to 1. At each step, if *n_dec* equals 1, i.e. the BDD is monolithic, control passes to procedure SPLIT (see section 4.1). This procedure evaluates the size of F_{dec} and if it exceeds *limit*, it decomposes it into subsets. *limit* is a parameter for the procedure and its value essentially depends on the image computation procedure complexity and on the set representation limit. SPLIT decomposes the original set into *n_dec* subsets returning each decomposition in F_{dec}. If *n_dec* is greater than 1, a previous decomposition has already been done and we move to the next step.

For each subset an image computation (function IMG) is called. Images of subsets are stored in T_{dec}. This allows the image computation procedure to deal with just one subset at a time decreasing BDD peak size.

After the image computation phase, a fixed point computation and other operations (see fig. 1) must be performed on the monolithic or the decomposed form. Functions COMPUTE_NEW, COMPUTE_REACHED, COMPUTE_FROM, and VERIFY_NEW are relatively simple and perform these operations on a conjunction–comparison base. These procedures can manage and return monolithic BDDs or decomposed ones, depending on the size of the BDD themselves. *n_dec* is set accordingly to the strategy used.

To deal with very large sets, we have implemented a quite efficient method to store and load BDDs from the main memory to the secondary one. The method requires a minimum of 4 bytes to store a single BDD node whereas the average occupation is of about 6 bytes per BDD node with BDDs with less than $500,000$ nodes. Every procedure is built to work indifferently with BDDs in the main or in the secondary memory depending on the general complexity and efficiency.

4 Set decomposition

Good decomposition of state sets is a key issue. As for each decomposition we have to compute $\textsc{Img}(f, C_1) + \textsc{Img}(f, C_2)$ instead of $\textsc{Img}(f, C)$ a good decomposition avoids re-computations.

Our decomposition technique is simple and fast and has almost no overhead in terms of nodes and a little cost in terms of CPU time. Given a BDD, the procedure tries to split it in two BDDs that represent subsets of the original set, and eventually recurs if the solution is still not optimal. The split is based on Boole's expansion theorem. By dynamically choosing the best splitting variable, we try to maintain overlaps among different decompositions low, avoiding repeated computations.

In the next subsections we describe the splitting routine (SPLIT) and the way we choose the best split variable (SELECT_VAR).

4.1 Procedure SPLIT

Given a BDD f, representing a set of states, we decompose it in BDDs, which represent subsets of the original set. We proceed trough a recursive splitting and during each step we split f in two BDDs g_1 and g_2.

Our procedure is essentially based on Boole's expansion, i.e. $f(x_1, x_2, \ldots, x_i, \ldots, x_n) = x_i \cdot f|_{x_i} + \overline{x_i} \cdot f|_{\overline{x_i}} = g_1 + g_2$. As the target is to decompose the BDD with minimal node overhead ($|g_1| + |g_2| \approx |f|$ [1]) but as, in general, we have to deal with large BDDs and also the time required is important, we have to select in a fast and efficient way the variable x_i to use. In other words we cannot compute all the cofactors with respect to all the variables, but we have to evaluate their size a priori.

The pseudo code of the main procedure is shown in fig. 3.

First, procedure SELECT_VAR (see section 4.2) selects the best splitting variable. Then the two cofactors, *pbdd_left* and *pbdd_right*, are computed. Finally the procedure tries to recur just in case the size of the BDDs obtained is still too large.

4.2 Procedure SELECT_VAR

This procedure returns the best splitting variable depending on the BDD structure. For each variable of the BDD *pbdd* calls COUNT and COST. The former (see fig. 5) estimates, due to the node sharing, the nodes of the positive or negative cofactor (depending on the *polarity*) of *pbdd* with respect to the i variable without computing the cofactors. It is quite fast, doesn't create any new BDD node and return a really good overestimation. The latter evaluates which variable is best for the split procedure, depending on the size of the left and the right cofactor.

We experimented with several cost functions. A split should produce balanced BDDs and with the smallest global increase in terms of BDD nodes. A good cost function minimizes the value $|(n\_n\_left - n\_n\_right)| + |((n\_n\_left + n\_n\_right) - n\_n)|$ where n_n is the number of nodes of the original BDD.

5 Experimental Results

We implemented the procedures presented above in an home–made tool written in C of about $15,000$ lines of code. BDD ordering heuristics are static, i.e. we don't

[1] If f is a BDD we denote with $|f|$ its number of nodes.

apply reordering heuristics during traversal. The number of BDD nodes is limited to $4,000,000$. We experimented with different traversal strategies [1], [3], [5], and [6]. Experiments ran on a 130 MIPS DEC Alpha with 256Mbyte of main memory. An effective use of secondary memory allows us to save relevant portions of BDDs and work with only a minimum number of BDD nodes in the main memory avoiding repeated page faults.

The columns of Tab. 1 give the name and the number of primary inputs (# PI), primary outputs (# PO), flip-flops (# FF), and gates (# G) for the MCNC, ISCAS'89, and ISCAS'89–addendum benchmarks.

In all the following tables Circuit indicates the name of the circuit, Level indicates the number of the level (intermediate or final) of the exact forward traversal, # Nodes is the number of nodes and # States is the number of reachable states. # Dec. indicates the number of decompositions, i.e. in how many images we decompose a single image step.

Tab. 2 shows data for sbc (MCNC), s1269 (ISCAS'89–addendum) and for s5378 (ISCAS'89).

In general, the overhead due to decomposing and recomposing sets is negligible with respect to traversal time. Thus we are able to decrease the peak node size, to simplify single image computation steps, and to lower the CPU time.

We reported data on sbc just to prove the feasibility of the approach also on small circuits. For this circuit the decomposition level is 2000 nodes, the maximum reached size is a bit larger than 4000 nodes, and we use 2 or 3 decompositions from level 4 on.

On circuit s1269, the standard approach explodes after level 2. Using the decomposed approach we are able to visit the circuit entirely. The decomposition procedure works quite well in this case as the BDD is quite "uniform". We are able to decompose the reached set at level 2, consisting of 16017 nodes, in 6 subsets, each having less that 3000 nodes. At level 3 a threshold of 7500 nodes is used, obtaining 12 partitions. Data show that levels 3 and 4 are critical, because of the intermediate computations. At level 4 the number of nodes of the BDD representation of the reachable state set decreases by two orders of magnitude and all other levels are quite simple to compute.

Circuit s5378 is even harder. The standard approach reaches level 2. The BDD of the reachable state set in this case is not particularly large, nevertheless it is quite difficult to decompose it in subsets, because of the large number of state variables (179). With our current implementation of decomposed traversal, not yet optimized, we reached level 3. We stopped because we estimated 100 hours were necessary to reach level 4. This amount of time is quite large, but, for the first time there are no virtual memory limits to the traversal procedure.

Tab. 3 collects data for the s1423 ISCAS'89 benchmark. Despite its relatively small size, s1423 is very difficult to handle during reachability analysis. Extensive experiments have been performed on this circuit [7], [4]. Though Ravi *et al.* [4] used the transition function, which is thought to be more powerful than the transition relation, they couldn't succeed in going beyond the 11^{th} level with a pure breadth–first traversal. To the best of our knowledge, nobody has yet presented data beyond that limit. Our standard approach, with a non-optimized traversal based on the transition relation, is able to deal with the circuit up to level 10. Afterwards, we resort to the decomposed approach. For this benchmark we have to decompose BDDs above approximately 60000 nodes. The degree of overlapping between different partitions is quite low because our decomposition techniques works quite well for this circuit, as, despite the relatively small number of state variables (74), the BDD sizes are enormous. We reached level 13; at that level the reachable state set has more than $4,000,000$ nodes and the monolithic representation is impossible to obtain with our memory limit.

6 Conclusions and Future Work

Symbolic FSM state space exploration techniques represent one of the major recent results of formal verification. Their limit resides in the inability to deal with large circuits. In this paper we propose a decomposed forward traversal that is exact and lowers BDD peak size during image computation and representation limits of the reachable state set. This technique has been proved also useful to avoid severe thrashing in the virtual memory system. Experimental results show that it works quite well with uniform and large sets.

We are currently working on a new traversal package built on top of the CUDD package [8]. The overall tool is $5 \div 10$ times or more faster than the tool we currently use. This is essentially due to three factors: the CUDD package is faster then the BDD package we have used so far, the reordering procedure built inside the CUDD package allows us to work with smaller BDDs, and new clustering and ordering techniques implemented in the traversal procedure allow more efficient traversals.

In particular the possibility to use different orders with different decompositions is quite attractive. We are also experimenting with new decomposition strategies trying to optimize the split procedure also for "non-uniform" BDDs.

References

[1] H. Touati, H. Savoj, B. Lin, R.K. Brayton, A. Sangiovanni-Vincentelli, "Implicit enumeration of finite state machines using BDDs," in *Proc. IEEE ICCAD'90*, November 1990, pp. 130–133

[2] H. Cho, G. Hachtel, S.W. Jeong, B. Plessier, E. Schwarz, F. Somenzi, "ATPG Aspects of FSM Verification," in *Proc. IEEE ICCAD'90*, November 1990, pp. 134–137

[3] G. Cabodi, P. Camurati, "Exploiting cofactoring for efficient FSM symbolic traversal based on the Transition Relation," in *Proc. IEEE ICCD'93*, October 1993, pp. 299–303

[4] K. Ravi, F. Somenzi, "High–Density Reachability Analysis," in *Proc. IEEE ICCAD'95*, November 1995, pp. 154–158

```
TRAVERSAL (δ, S_0)
{
R = F = N = S_0;
while (N ≠ ∅)
    {
    T = IMG (δ, F);
    N = T · ¬R;
    R = R + N;
    F = BEST_BDD (N, R);
    }
return (R);
}
```

Figure 1: Exact Symbolic Forward Traversal.

```
DECOMPOSED_TRAVERSAL (δ, S_0, limit)
{
n_dec = 1;
R_dec = F_dec = N_dec = S_0;
continue = true;
while (continue)
    {
    if (n_dec == 1)
        n_dec = SPLIT (F_dec, limit, n_dec);
    for (i = 0; i < n_dec; i++)
        T_dec [i] = IMG (δ, F_dec [i]);
    N_dec = COMPUTE_NEW (T_dec, R_dec, limit, n_dec);
    R_dec = COMPUTE_REACHED (R_dec, N_dec, limit, n_dec);
    F_dec = COMPUTE_FROM (N_dec, R_dec, limit, n_dec);
    continue = VERIFY_NEW (N_dec, n_dec);
    }
return;
}
```

Figure 2: Exact Decomposed Forward Traversal.

[5] O. Coudert, J.C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits" in *Proc. IEEE ICCAD'90*, November 1990, pp. 126–129

[6] R.K. Ranjan, A. Aziz, R.K. Brayton, B. Plessier, C. Pixley, "Efficient BDD Algorithms for FSM Synthesis and Verification," IWLS'95: IEEE International Workshop on Logic Synthesis, Lake Tahoe, CA, USA, May 1995

[7] H. Cho, G.D. Hatchel, E. Macii, M. Poncino, K. Ravi, F. Somenzi, "Approximate Finite State Machine Traversal: Extensions and New Results," IWLS'95: IEEE International Workshop on Logic Synthesis, Lake Tahoe, CA, USA, May 1995, pp. 3.1–3.5

[8] F. Somenzi, "CUDD: CU Decision Diagram Package – Release 1.0.4," Technical Report, Dept. of Electrical and Computer Engineering, University of California, Boulder, November 1995


```
SPLIT (pbdd, limit, n_dec)
{
if (|pbdd| > limit)
    {
    var = SELECT_VAR (pbdd);
    if (var > 0)
        {
        pbdd_left = var · pbdd;
        pbdd_right = ¬var · pbdd;
        n_dec++;
        SPLIT (pbdd_left, limit, n_dec);
        SPLIT (pbdd_right, limit, n_dec);
        }
    }
return (n_dec);
}
```

Figure 3: Splitting a BDD.

```
SELECT_VAR (pbdd)
{
var = -1;
for (i = 0; i < VAR_NUMBER (pbdd); i++)
    {
    n_n_left = COUNT (pbdd, VAR (i), 1, 0);
    n_n_right = COUNT (pbdd, VAR (i), 0, 0);
    if (COST (|pbdd|, n_n_left, n_n_right))
        var = i;
    }
return (var);
}
```

Figure 4: Selecting the best split variable.

```
COUNT (pbdd, var, polarity, n_n)
{
if (TERMINAL_CASE (pbdd))
    return (n_n);
if (pbdd.count == 0)
    {
    if (pbdd.var == var )
        if (polarity == 1)
            n_n = COUNT (pbdd.left, var, polarity, n_n);
        else
            n_n = COUNT (pbdd.right, var, polarity, n_n);
    else
        {
        n_n = COUNT (pbdd.left, var, polarity, n_n);
        n_n = COUNT (pbdd.right, var, polarity, n_n);
        }
    n_n++;
    pbdd.count = n_n;
    }
return (n_n);
}
```

Figure 5: Counting the Number of Nodes.

Circuit	# PI	# PO	# FF	# G
sbc	40	56	28	1011
s1423	17	5	74	657
s5378	35	49	179	2779
s1269	18	10	37	560

Table 1: Example statistics for some MCNC, ISCAS'89, and ISCAS'89–addendum benchmarks.

Circuit	Level	Reached		Monolithic Approach	Decomposed Approach	
		# Nodes	# States	CPU Time	# Dec.	CPU Time
sbc	10	4277	$1.54592 \cdot 10^{5}$	86.1	3	82.3
s1269	1	481	$4.340 \cdot 10^{3}$	0.2	=	0.2
	2	16017	$1.308 \cdot 10^{7}$	6.7	=	6.7
	3	103522	$8.034 \cdot 10^{8}$	ovf	6	1145
	4	805	$8.842 \cdot 10^{8}$	-	12	4971
	5	804	$9.309 \cdot 10^{8}$	-	=	4975
	6	805	$9.776 \cdot 10^{8}$	-	=	4978
	7	803	$1.024 \cdot 10^{9}$	-	=	4980
	8	813	$1.066 \cdot 10^{9}$	-	=	4982
	9	803	$1.131 \cdot 10^{9}$	-	=	4983
	10	803	$1.131 \cdot 10^{9}$	-	=	4983
s5378	1	539	$1.049 \cdot 10^{6}$	1.4	=	1.4
	2	10395	$1.274 \cdot 10^{9}$	50.5	=	50.5
	3	31349	$1.729 \cdot 10^{12}$	-	20	17^{h}

Table 2: Comparison between Standard and Decomposed Traversal on large circuits. = means that no decomposition was necessary; ovf indicates overflow on BDD nodes; - means no data available.

Level	# Dec.	Peak size BDD	# Reached Nodes	# Reached States	CPU Time
1	1	149	205	$5.450 \cdot 10^{2}$	0.7
2	1	272	372	$3.345 \cdot 10^{3}$	1.5
3	1	836	785	$5.557 \cdot 10^{4}$	2.0
4	1	2269	1625	$3.922 \cdot 10^{5}$	3.9
5	1	7523	3228	$2.080 \cdot 10^{6}$	7.5
6	1	15691	6300	$8.493 \cdot 10^{6}$	12.8
7	1	35695	13350	$3.370 \cdot 10^{7}$	25.9
8	1	106089	29465	$1.111 \cdot 10^{8}$	80.3
9	1	268527	72562	$4.896 \cdot 10^{9}$	295.5
10	1	721179	194036	$1.683 \cdot 10^{9}$	998.4
11	4	874074	604998	$7.990 \cdot 10^{9}$	4997.5
12	15	1348074	1870790	$2.303 \cdot 10^{10}$	5^{h}
13	53	1257884	5968310^{*}	$7.952 \cdot 10^{10}$	57^{h}

Table 3: Decomposed Traversal on s1423 circuit. * means that in this case the monolithic representation doesn't exist and then the number is overestimated on the decomposed representation.