Free MDD-Based Software Optimization Techniques for Embedded Systems

Chunghee Kim
Research Institute of Eng. and Tech.,
Hanyang Univ., KOREA
(Visiting Researcher at U. C. Berkeley)

Luciano Lavagno
DIEGM, Universita' di Udine
Via delle Scienze 208
I-33100 Udine, ITALY

Alberto Sangiovanni-Vincentelli
Dept. of EECS, U. C. Berkeley
Berkeley, CA94720

Abstract

Embedded systems make a heavy use of software to perform Real-Time embedded control tasks. Embedded software is characterized by a relatively long lifetime and by tight cost, performance and safety constraints. Several super-optimization techniques for embedded softwares based on Multi-valued Decision Diagram (MDD) representations have been described in the literature, but they all share the same basic limitation. They are based on standard Ordered MDD (OMDD) packages, and hence require a fixed order of evaluation for the MDD variables on every execution path. Free MDDs (FMDDs) lift this limitation, and hence open up more optimization opportunities. Finding the optimal variable ordering for FMDDs is a very difficult problem. Hence in this paper we describe a heuristic procedure that performs well in practice, and is based on FMDD cost estimation applied to recursive cofactoring. Experimental results show that our new variable ordering method obtains often smaller embedded software than previous (sifting-based) methods.

1 Introduction

Embedded systems include hardware and software components operating together to achieve a common goal. Real-time embedded systems are often characterized by the need for running several tasks on a limited set of processing units and react continuously to their environment at the speed of the environment [1, 2]. These systems are widely used in vehicle control, consumer electronics, communication systems and so on.

Design of an embedded system includes design specification, validation and hardware and software synthesis. Embedded software is characterized by tight cost, performance and real-time constraints. For this reason, embedded software design makes heavy use of Finite State Machine-like specifications, and requires better compiler optimization algorithms than general-purpose computing.

Binary Decision Diagrams (BDDs) and Multi-valued Decision Diagrams (MDDs) (a generalization of BDDs) have shown to be a convenient notation to represent and optimize decision-intensive embedded software modules, since they are a representation for Finite State Machines (FSMs) that is quite close to the execution model on an embedded processor. Several super-optimization techniques for FSMs based on MDD representations have been described in the literature [1, 3, 4]. However, they all share the same basic limitation. They are based on standard Ordered MDD (OMDD) packages, and hence require a fixed order of evaluation for the MDD variables.

In this research, we have developed a software optimization technique for embedded softwares based on the use of Free Multi-valued Decision Diagrams (FMDDs) [5]. An MDD is called FMDD if each variable is tested at most once on each path. So an OMDD is an FMDD with the property that on each path the avriables are tested in a predefined order. It was shown that FMDDs provide a canonical representation and allow effective solutions of the basic tasks in Boolean manipulation similarly as the well known OMDDs do [5, 6].

We evaluated our algorithms by embedding them in the publicly available tool POLIS [1]. The POLIS system is intended for control dominated embedded systems. It uses a system specification based on a network of extended asynchronous FSMs called Co-design Finite State Machines (CFSM). It currently synthesizes software from a CFSM using an OMDD as an intermediate representation. Moreover, it offers software size and performance estimation capabilities.

All known MDD-based software synthesis approaches are based on heuristically minimizing the size of the MDD, because for control-dominated applications[1] it tracks well the size of the final software implementation. It is well-known that the size of an MDD representation of a function (in this case the sequential function relating embedded software inputs and outputs) depends heavily on the order of the variables. Unfortunately, finding an optimum variable ordering even for the sub-class of OMDDs is well known to be a co-NP-complete problem [7].

Of course, things become even worse when one considers the broader class of FMDDs. This requires the introduction of heuristic techniques to determine the best variable order along every computation path. In this paper we present such a heuristics, based on the key observation that *in an FMDD graph recombination near the leaves is much less frequent than in an OMDD*, essentially due to the different orders that make sharing much more difficult. Hence we propose an efficient FMDD size estimation technique, given a partial variable ordering starting from the root, that conservatively assumes *no recombination* when estimating the size of the portion still to be built.

We show experimentally that our heuristics performs better than OMDD-based techniques with an acceptable compilation time. We also implemented an exact (but very expensive) ordering algorithm and we show that the heuristic technique gives identical results on small examples, where both techniques are applicable.

The organization of this paper is as follows. The POLIS design process and previous work are briefly discussed in Section 2. The new variable ordering technique is described in Section 3. Experimental results and conclusions are given in Sections 4 and 5.

[1] I.e., applications in which the complexity of the decision process that determines the outputs of the embedded system dominates over the numerical computations.

```
-1 BEGIN:
 1 ASSIGN: o_st=2
 2 TEST: *RESET==1
   3 ASSIGN: o_WHEEL_PULSES=0
   4 ASSIGN: o_OC3_START=1024
   5 ASSIGN: *o_WHEEL_PULSES=1
   6 ASSIGN: o_counter=0
   7 ASSIGN: *o_OC3_START=1
   0 END
 2 TEST: *RESET==0
   8 TEST: _st==2
     9 TEST: *OC3_END==1
       10 ASSIGN: o_WHEEL_PULSES=counter
       goto 4
     9 TEST: *OC3_END==0
       11 TEST: *WHEEL_PULSE==1
         12 ASSIGN: o_counter=counter+1
         goto 0
       11 TEST: *WHEEL_PULSE==0
         goto 0
   8 TEST: _st==1

     goto 3
```

Figure 1: An s-graph example.

2 Previous work

2.1 Software Synthesis in POLIS

In the POLIS design process [1], an abstract design description in the form of communicating CFSMs is partitioned and implemented into software and hardware. The CFSMs are extended Finite State Machines[2] communicating asynchronously by means of buffers. The internal representation of software in POLIS is a restricted Control-Data Flow Graph (CDFG) called s-graph. It is an extended decision diagram (Directed Acyclic Graph) with a source (BEGIN), a sink (END), and two types of internal nodes. A TEST node evaluates an expression depending on the CFSM input and state variables and branches to one of several child nodes depending on the result. An ASSIGN node evaluates a function and assigns its result to a CFSM output or state variable.

Figure 1 shows a simple s-graph example, describing part of a dashboard controller. TEST nodes (e.g., the TEST node number 2, checking if input *RESET has value '1' or '0') are represented by two rows, identified by the same node number. The CFSM represented by this s-graph computes the number of events on input *WHEEL_PULSE that occur between two events on input *OC3_END[3]. It uses state variables "_st" (with symbolic values '1' and '2') and "counter" (an integer). Event *OC3_END arrives periodically[4], and its occurrences are accumulated in state variable "counter". The CFSM outputs the current value of "counter" on output o_WHEEL_PULSES, and signals this fact by assigning '1' to the corresponding presence bit *o_WHEEL_PULSES, every time *OC3_END arrives.

An algorithm taking an OMDD representation of the transition function of the CFSM and obtaining an s-graph from it is described in [1]. Assuming that functions computed by TEST and ASSIGN nodes have almost the same execution time and size, the algorithm minimizes heuristically:

- execution time, since each function on which the CFSM output depends is evaluated at most once for each execution of a CFSM transition,
- code size, since code synthesis heuristically looks for the ordering of evaluation that yields a small MDD size[5].

After the s-graph is constructed and optimized, it is translated into a C function. A TEST node is translated into an "if" or "switch" statement, and an ASSIGN node is translated into an assignment statement. The generated C program has the same structure as its original s-graph [8].

2.2 MDD Minimization

The Ordered Binary Decision Diagram (OBDD) [7] and its generalization to multi-valued variables (OMDD) have proven to be useful in many applications as an efficient data structure for representing and manipulating Boolean functions. BDDs and MDDs have been used in a wide variety of CAD applications including logic synthesis and verification.

Since finding the best variable order is co-NP complete, several heuristics have been developed. Three kinds of approaches have been used to find a good variable order.

1. Methods based on the circuit topology [9, 10, 11, 12]:
   Several heuristics have been developed and have been shown to be practical in real circuits. These methods are based on the several network topologies [9, 10, 12] and presence of a don't care sets [11].
2. Iterative improvement methods [13, 14]:
   This approach is based on variable exchange from the initial variable order and is effective (generally better than topology-based methods), but the results depend on the initial variable orders.
3. Exhaustive methods based on branch and bound [15, 16]:
   The key part of these algorithms is a lower bound technique to find optimum solutions. But, these exact methods are so far applicable to the functions with less than 20 variables and more than 20 in special cases.

In many cases, including software synthesis, FMDD-based techniques can be superior than OMDD-based techniques. In contrast to OMDDs, FMDDs allow more efficient representations of Boolean functions. For example, FHS-function, indirect storage access function, and hidden weighted bit function can be represented by quadratic size FMDDs while OMDD-representations of these functions are of exponential size [6].

Figure 2 shows two possible implementations of a portion of an s-graph. In the top half, the order of the two variables OR_43 and AND_38 is different on each path in the FMDD. In the bottom half, even the best OMDD ordering yields a larger s-graph, due to the need to keep the same ordering on all paths.

Unfortunately, most of the work on variable ordering has focused on the simpler problem for OMDDs. For example, even though [6] proposed an FBDD-based data structure for Boolean manipulation, its variable ordering technique was based on a modification of one developed for OBDDs.

3 Variable Ordering for Free MDDs

In this work we developed a new variable ordering method for FMDDs, based on a top-down recursive paradigm followed by a sub-graph merging step. The approach is based

[2] FSMs with transition and output functions operating on integer variables using arithmetic and relational operators.

[3] Variables with a '*' denote event presence when they have a value of 1, while variables without a '*' denote event values.

[4] It is generated by a timer CFSM that receives as input event *o_OC3_START and waits for 1024 clock cycles before producing *OC3_END.

[5] Generally this is based on algorithms such as sifting [14] that are not exact, but yield very small MDDs in a reasonable amount of compilation time.

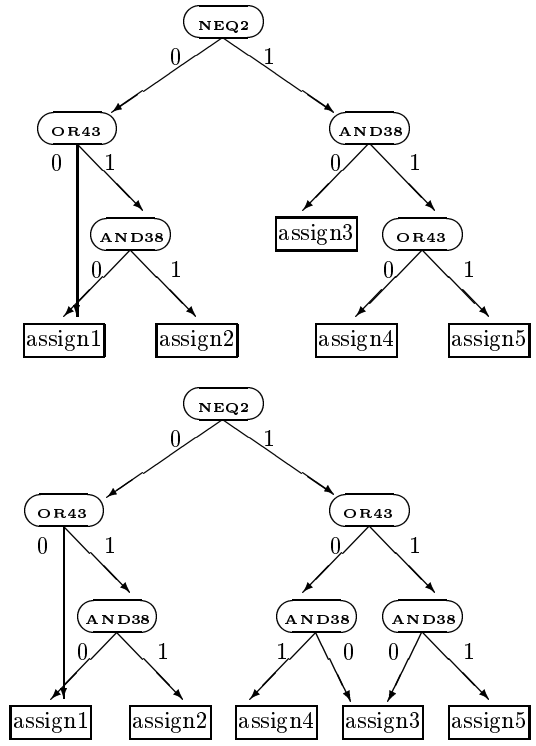

Figure 2: Example of an FMDD and OMDD.

on a heuristic estimation of the size of the MDD obtained by selecting a new variable for the current recursion level, assuming that no sub-graph merging will occur further down.

The new algorithm will be illustrated by using the example shown in Figure 1. The OMDD order of the four input variables is *RESET, _st, *OC3_END and *WHEEL_PULSE[6]. In this specific example all input variables are tested before any output variable is assigned, but in general both the original algorithm of [1] and our new algorithm can interleave inputs and outputs (as long as an output appears after its support [1]).

3.1 Nodes Clustering

In order to reduce the complexity of the problem, we heuristically merge several ASSIGN nodes executed under the same set of input conditions into a single *cluster*. For example, the ASSIGN nodes in Figure 1 are merged into the following five clusters:

```
Cluster1 : o_st=2
Cluster2 : o_WHEEL_PULSES=0
Cluster3 : o_OC3_START=1024
           *o_WHEEL_PULSES=1
           o_counter=0
           *o_OC3_START=1
Cluster4 : o_WHEEL_PULSES=counter
Cluster5 : o_counter=counter+1
```

Note that the string of ASSIGN nodes 3 to 7 in Figure 1 is split into two clusters, due to the sub-graph sharing from node 4 (its predecessors are nodes 3 and 10). This essentially means that our heuristics will *never increase the amount of ASSIGN node sharing* with respect to the initial s-graph produced by POLIS. But the main advantage of FMDDs is the ability to generate "tree-like" s-graphs (without much recombination) that are, as we will show, generally smaller than the s-graphs generated from OMDDs.

In the rest of the paper we will represent the transition function before the construction of the FMDD in the form of a two-level *execution table*, that associates a cluster number with an input cube. An input cube is an assignment of a value or a '-' (meaning don't care) to the input variables of the CFSM. For example, the ASSIGN statements merged into Cluster 3 are executed by the CFSM under the following conditions (disjoint input cubes) on the values of variables *RESET, _st, *OC3_END and *WHEEL_PULSE: either "1---", or "0 2 1-", or "0 1--".

On the other hand, Cluster 1 is executed no matter what is the value of the input variables[7]. This means that it can be executed *immediately* (as the first node of the s-graph after the BEGIN node), without examining any input variable. This corresponds to choosing output variable o_st as the first variable in the FMDD order.

3.2 Variable Ordering

Figure 3(a) shows the execution table for the example shown in Figure 1 after Cluster 1 has been chosen as the first node of the s-graph.

Our recursive procedure operates on an execution table as follows.

1. Minimize (using standard multi-valued two-level minimization techniques [17]) the current execution table.
2. If there exists a line with all input don't cares, the corresponding output Cluster can be added to the s-graph, and the procedure is called again after removing the rows corresponding to this Cluster.
3. Otherwise, if there exists an input variable that is the only care column for some output clusters, then this variable is selected. The execution table is split in as many sub-tables as the number of values of the variable (by *cofactoring* it with respect to the variable), and the procedure is repeated recursively.
4. Otherwise, the input variable *with the minimum estimated cost* is selected (see below for a more precise description of this cost function). The execution table is split in as many sub-tables as the number of values of the variable (by *cofactoring* it with respect to the variable), and the procedure is repeated recursively.

Step 4 is the key to our heuristic procedure. Output Clusters cannot be chosen at this step, because there is not enough information in the variables tested so far in the recursion to determine if they can be executed or not (their support has not been selected yet). So we used a cost based variable selection method.

Our heuristic sorts variables according to the number of fewer don't cares and then slightly modifies the sorted variable order to search a large solution space.

This variable ordering technique based on variables sorting is reasonable because the size of a generated s-graph depends on the number of sub-execution tables. If we chose an input variable with many don't cares in the corresponding execution table column, this would cause many rows to be duplicated in the cofactors. Since there is little recombination in

[6]Variables *RESET, *OC3_END and *WHEEL_PULSE are binary-valued, and variable _st has symbolic values '1' or '2'.

[7]The CFSM performs an initial transition from state 1 to state 2 and remains in state 2 forever.

	a	b	c	d
2	1	–	–	–
2	0	1	–	–
3	1	–	–	–
3	0	2	1	–
3	0	1	–	–
4	0	2	1	–
5	0	2	0	1

$a = *RESET$
$b = \_st$
$c = *OC3\_END$
$d = *WHEEL\_PULSE$

(a) An execution table for the clusters in Figure 1.

	b	c	d
2	1	–	–
3	2	1	–
3	1	–	–
4	2	1	–
5	2	0	1

$for\ a = 0$

(b) A divided sub-table for $a = 0$ when variable a is selected.

	a	b	d
2	1	–	–
2	0	1	–
3	1	–	–
3	0	1	–
5	0	2	1

$for\ c = 0$

	a	b	d
2	1	–	–
2	0	1	–
3	–	–	–
4	0	2	–

$for\ c = 1$

(c) Divided sub-tables for $c = 0$ and $c = 1$ when variable c is selected.

Figure 3: An execution table and variable selections.

Examples	Rand	Polis		New	
	Size (nodes)	Size (nodes)	CPU (sec)	Size (nodes)	CPU (sec)
belt1	149	61	0.1	57	0.1
eng_cross_display	149	93	0.1	83	0.1
spd_cross_display	145	91	0.1	79	0.1
controller	107	80	0.1	65	0.1
arbiter	238	106	0.1	112	5.0
bat_diag	106	55	0.1	54	0.1
curve	898	214	2.0	211	15.0
curve_front	898	214	2.0	211	15.0
curve_rear	1322	300	6.0	255	52.0
driver	973	181	1.0	187	86.0
long_acc_der	47	44	0.1	43	0.1
long_spd_cal_trs	119	86	0.1	85	0.1
long_spd_cal_val	190	99	0.1	89	0.1
long_spd_diag_acc	530	86	1.0	86	18.0
long_spd_diag_par	161	40	0.1	41	0.1
long_spd_strat	231	174	0.1	66	1.0
mot_ctrl_damage	318	104	0.1	104	10.0
speed_sens	74	46	0.1	45	0.1
steer_ang_cal_init	704	170	1.0	122	24.0
steer_ang_cal_val	1716	202	1.0	193	59.0
steer_ang_corr	651	167	2.0	176	19.0
steer_ang_diag	263	93	0.1	101	4.0
steer_speed_cal	186	76	0.1	76	0.1
steer_wheel	807	185	1.0	158	12.0
ver_acc_diag	137	54	0.1	52	0.1
belt2	127	46	0.1	38	0.1
cross_display	113	80	0.1	69	0.1
fuel	126	52	0.1	50	0.1
timer1	4593	97	4.0	97	11.0
belt_controller	73	48	0.1	48	0.1
timer2	70	40	0.1	39	0.1
Total	16221	3384	23.1	3092	332.7

Table 1: Comparison with POLIS (the five groups of examples are "DAC_DEMO", "TLC", "MCA200", "DASHBOARD" and "BELT").

FMDDs, this duplication will (most likely) not be recovered later.

Let us consider now more precisely the cost function used to sort variables on the current execution table. We sort variables according to the following cost:

$$cost_{variable\_sort}(v) \ = \ \sum_f q_{f,v}$$

where f ranges over all the cubes of the table. The value of $q_{f,v}$ is 1 if the variable has a defined value in the cube, and the number of values of the variable otherwise. As mentioned above, this heuristically estimates the number of copies of the cube generated if v is selected for the recursion. For example in Figure 3(b), $cost_{variable\_sort}(b) = 5$ and $cost_{variable\_sort}(c) = 7$.

After variable sorting, we use a heuristic technique to improve the top variable choice, based on an idea similar to sifting. We move each variable in the order within a specified range (usually from 5 to 10, in order to keep an acceptable execution time), and for each order we recursively decompose the current execution table and estimate its FMDD size. The order with the minimum cost is saved and the first variable in this order is selected.

The size cost $cost_{size\_estimation}(g_r)$ for an s-graph g_r with variable order r is estimated as follows.

$$cost_{size\_estimation}(g_r) \ = \ |TEST\ nodes\ in\ g_r| + |ASSIGN\ nodes\ in\ g_r|$$

The size of TEST and ASSIGN nodes in g_r is computed using the techniques in [1]. The number of test nodes is computed from the number of sub-execution tables and the number of assign nodes can be calculated during the recursive execution-table division.

If a variable is selected, two or more sub-execution tables will be generated according to the possible variable values. If a selected variable can have values '0', '1' or '2', three sub-execution tables have to be generated (one for each value).

For example, if we select variable a in Figure 3(a), we have two recursions (sub-cases):

- for a='1' we can immediately select and execute Clusters 2 and 3.
- for a='0' we generate the table shown in Figure 3(b).

If variable c had been selected in Figure 3(a), two sub-execution tables would be generated for c='0' and c='1', as shown in Figure 3(c). In the first recursion, variable a would be selected next. In the second recursion, Cluster 3 and variable a would be selected next.

At the end of the recursive procedure, we obtain an FMDD that is a tree. Hence we try to merge isomorphic sub-graphs as much as possible, starting from the leaves [7].

4 Experimental Results

The new variable ordering algorithm for s-graph optimization has been implemented in C on a DEC5000 workstation. We have tested our algorithm on several realistic embedded system examples available on the POLIS web site [18]. The size shown in all the tables is the number of s-graph nodes, that is known to correlate well with code size. The code execution time on the target processor is practically not affected by the new optimization procedure, since both the old and the new technique test each input and assign each output at most once along every execution path. More precise data on actual code size and execution time will be provided in the final version.

The first column gives the name of the CFSM. We used five groups of examples which are "DAC_DEMO", "TLC", "MCA200", "DASHBOARD" and "BELT" on [18]. The second column gives the size of the generated s-graph, obtained with the random variable order provided in the POLIS input file.

The third column gives the s-graph size as obtained by POLIS with the "sift" algorithm [14] (it sifts one variable at a time up and down and freezes it in the position where the MDD size is minimized). The fifth column gives the s-graph size as obtained by our new algorithm. These results show that the size of the s-graph generated by our algorithm with Free MDDs is smaller than that generated by using ordered BDDs. Note how the results of our new algorithm are especially better on some large s-graphs (e.g., curve_rear, long_spd_strat). Our FMDD construction method is just a heuristics, and hence it is not automatically guaranteed to obtain better results than OMDD-based ones (see, e.g., the CFSM "arbiter").

The fourth and sixth columns show the s-graph generation times (in seconds) required by the two methods. Finding the variable ordering and building the s-graph takes only a few CPU seconds for most of the examples (52 seconds on the largest example, curve_rear). Our clustering method is also effective in order to reduce the execution time without increasing the s-graph size. For example, the "curve_rear" example would take 163 CPU seconds without clustering.

For the sake of comparison, we also implemented an *exact* FMDD variable ordering algorithm, based on exhaustive search. It could be applied to only the smallest CFSMs of Table 1, namely long_acc_der, long_spd_cal_val, speed_sens, steer_speed_cal, ver_acc_diag. *In all these cases the cost of the heuristic and exact solution are exactly the same.*

5 Conclusions

We have described a new variable ordering technique for MDD-based software synthesis from Codesign Finite State Machines. The technique uses Free MDDs, that do not constrain the variable ordering along every execution path to be the same, in order to achieve near-optimum code size with maximum execution speed for a given CFSM specification. It is based on a heuristic cost measure that minimizes the number of row splittings in a tabular representation of the CFSM transition function.

Our algorithm has been implemented and applied to Real-Time reactive embedded system examples. The experimental results show that the new algorithm is more effective than the previous algorithm, in particular when applied to larger examples. It also achieves optimum results on small examples to which exact methods can be applied.

In the future, we are planning to develop a more accurate cost estimation procedure for each node during variable selection. We will also try to use our technique in other application areas of FMDDs, such as formal verification, pass transistor synthesis [19], and asynchronous circuit synthesis [20].

References

[1] F. Balarin, E. Sentovich, et. al., Hardware-Software Co-Design of Embedded Systems - The POLIS approach, Kluwer Academic Publishers, 1997.

[2] S. Edwards, L. Lavagno, et. al., "Design of Embedded Systems: Formal Models, Validation, and Synthesis," Proc. of the IEEE, Vol. 85, No. 3, p. 366, March 1997.

[3] P. McGeer, K. McMillan, et. al., "Fast Discrete Function Evaluation using Decision Diagrams," Proc. of ICCAD, pp. 402-407, 1995.

[4] P. Ashar and S. Malik, "Fast Functional Simulation using Branching Programs," Proc. of ICCAD, pp. 408-412, 1995.

[5] J. Gergov and C. Meinel, "Efficient Boolean Manipulation With OBDD's can be Extended to FBDD's," IEEE Trans. on Computers, Vol. 43, No. 10, pp. 1197-1209, Oct. 1994.

[6] J. Bern, J. Gergov et. al., "Boolean Manipulation with Free BDD's: First Experimental Results," Proc. of the European Design and Test Conference., pp. 200-207, 1994.

[7] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE Trans. Computers, Vol. C-35, No. 8, pp. 677-691, 1986.

[8] M. Chiodo, P. Giusto, et. al., "Synthesis of Software Programs for Embedded Control Applications," Proc. of ACM/IEEE Design Automation Conference, pp. 587-592, 1995.

[9] S. Malik, A. Wang, et. al., "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," Proc. of ICCAD, pp. 6-8, 1988.

[10] M. Fujita, H. Fujisawa and Y. Matsunaga, "Variable Ordering Algorithms for Ordered Binary Decision Diagrams and Their Evaluation," IEEE Trans. on Computer-Aided Design, Vol. 12, pp. 6-12, Jan. 1993.

[11] Y. Hong, P. Beerela et. al., "Don't card-based BDD Minimization for Embedded Software," Proc. of ACM/IEEE 35th Design Automation Conference, pp. 506-509, 1998.

[12] H. Fujii, G. Ootomo and C. Hori, "Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams," Proc. of ICCAD, pp. 38-41, 1993.

[13] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of Binary Decision Diagrams Based on Exchanges of Variables," Proc. of ICCAD. pp. 472-475, 1991.

[14] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," Proc. of ICCAD, pp. 42-47, 1993.

[15] S. Friedman and K. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams", IEEE Trans. on Comput., Vol. C-39, No. 5, pp. 710-713, May 1990.

[16] R. Drechsler, N. Drechsler and W. Gunther, "Fast Exact Minimization of BDDs," Proc. of ACM/IEEE 35th Design Automation Conference, pp. 200-205, 1998.

[17] R. Rudell and A. Sangiovanni-Vincentelli, "ESPRESSO-MV: Algorithms for Multiple-Valued Logic Minimization," Proc. of Custom Integrated Circuits Conference (CICC), pp. 230-234, 1985.

[18] See http://www-cad.eecs.berkeley.edu/~polis/.

[19] M. Tachibana, "Synthesize pass transistor logic gate by using free binary decision diagram," Proc. of Tenth Annual IEEE International ASIC Conference and Exhibit, New York, NY, 1997.

[20] K.Y. Yun, B. Lin, D. Dill, S. Devadas, "BDD-based synthesis of extended burst-mode controllers," IEEE Transactions on Computer-Aided Design, Vol.17, No. 9, pp. 782-792, Sep. 1998.