Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 6 (2): 297-302, 2006
ISSN 1812-5654

Slicing Floorplan Using BDD-based Constraint Solver

Liu Chun-Chen
Department of Electrical Engineering, University of Southern California,
Los Angels California, USA 90089-0911

Abstract: Here we use the CUDD package as our main method to solve the floorplan problem. After surveying some of the functions and implementation of the CUDD package, we choose the functions in the basic CUDD library instead of the procedure library provided by nanotrov, which we need to get the parser and do much more complicated implementation to transform .blif files into the form we want. We use the fast construction of BDD provided by CUDD package and trace them using the test cases, which we generate all possible solutions and then feed into the BDD to find the feasible ones. Since we do the exhaustive search, we confirm that we can get the optimal solution. We implement two project flows, one with exhaustive search and the other with random generation. In the flow 1 we can get the optimum solutions successfully, but the constraints, memory usage and running time, of the CUDD limit the number of modules which we can handle. In the flow 2 we can get the local optimum solutions successfully, but the quality of solutions are depended on the random solutions.

Key words: Soc, verification, BDD, CUDD

INTRODUCTION

Slicing floorplan structure: Before introducing the slicing tree structure, we first introduce the rectangular dissection problem.

A rectangular dissection is to sub-divide a given rectangle by a finite number of horizontal and vertical line segments into a finite number of non-overlapping rectangles. These rectangles are named basic rectangles. One can divide the rectangle by any means he or she wants[1,2].

A slicing structure, rather than slicing tree structure, is a rectangle dissection that can be obtained by repetitively subdividing rectangles horizontally or vertically. The Fig. 1 show the basic concept of what a slicing structure is like.

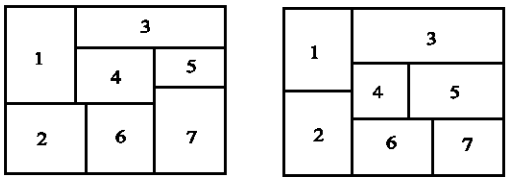


Fig. 1: The organized slicing structure. Floorplanning technology is a method to place all modules into a chip area. Modules: 1, 2,3,4,5,6,7

A slicing structure can be modeled by a binary tree with n leaves and n-1 internal nodes, where each node represents a vertical line or the horizontal cut line. The leaves, on the other hand, represent the basic rectangles. This binary tree is thus call a slicing floorplan tree (slicing tree for short)[3,4], or the slicing tree structure. There is one thing important which we want to clarify that a slicing structure can be modeled (or mapped) by more than one slicing tree--- that is to say, a slicing structure can have one or more slicing trees to represent itself. The Fig. 2a shows this condition--- two slicing trees for the same slicing structure.

Here we make some explanation about the slicing tree. Take Fig. 2b for example:

1. The left child of the H node means that it lies under the horizontal cut line, while the right child represents the basic rectangle above the horizontal cut line.
2. The left child of the V node means that it is at the left side of the vertical cut line, while the right child represents the basic rectangle on the right side of the vertical cut line.

To solve this problem, we introduced the concept of skewed slicing tree. A skewed slicing tree is a slicing tree that no node and its right child are the same. Take the above graph for example, Fig. 2c is not a skewed slicing tree for the reason that the internal node H in the second level and its right child are the same----- they are both H.

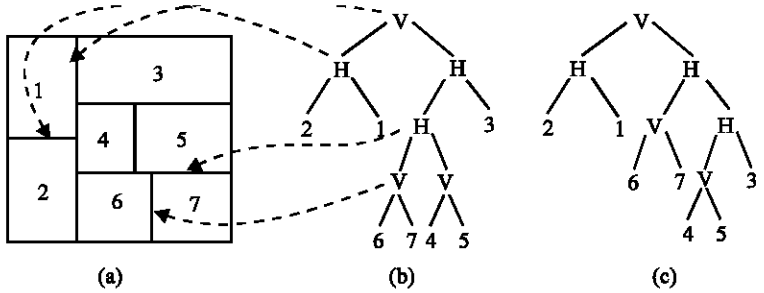


Fig. 2: a) Slicing floorplan, b) A slicing tree (skewed) and c) another slicing tree (non-skewed)

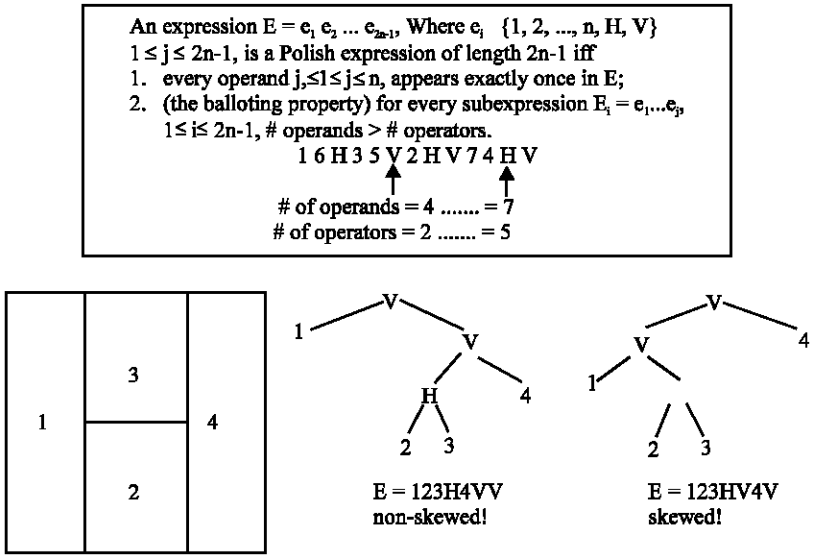


Fig. 3: The polish expression

Once we have found the skewed slicing tree, there is a one-to-one mapping between a slicing structure and its skewed slicing tree.

Polish expression: A polish expression can be derived by a method called post-order traversal[5,6] applying to the skewed slicing tree. Since we do the traversing of the whole tree, the expression has the same length as the total number of nodes in the slicing tree, that is, 2n-1(n is the number of rectangles). The polish expression has two properties, they are given in the Fig. 3.

Normalized polish expression has two constraints.(a)constraint 1: for every subexpression Ei = e1 … ei, 1 = i = 2n-1, # operands > # operators (b)constraint 2: no consecutive operators of the same type (H or V) and result will become H 1 6 3 5 V 2 V 7 4 H.

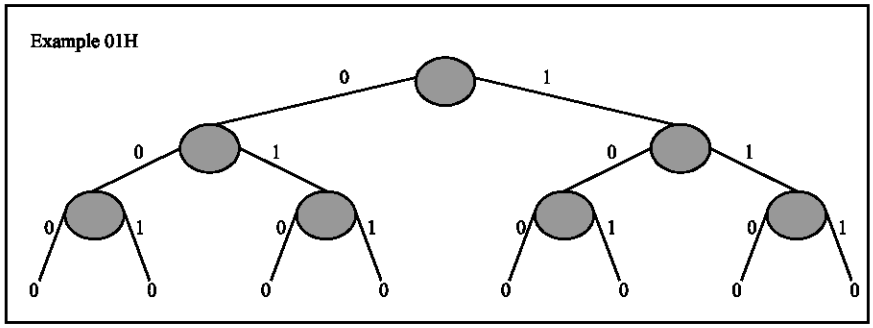
By doing post-order traversal we can derive a polish expression[7]. Unfortunately, the same problem arises as the slicing tree does----- a slicing structure can have more than one slicing trees, while each of the slicing trees has their own polish expression. Therefore, as the slicing tree do, we need a well defined expression structure called normalized polish expression. A polish expression is called normalized if it has no consecutive operators of the same type (V or H). According to the lemma developed, we claim that there is a one-to-one mapping between skewed slicing tree and normalized polish expression. As we stated earlier, there is a one-to-one mapping between a slicing structure and its skewed slicing tree. Thus we can conclude that there is a one-to-one mapping between a slicing structure and its normalized polish expression. We are going to use the normalized polish expression as our solution to the slicing structure[8,9].

PROBLEM FORMULATION

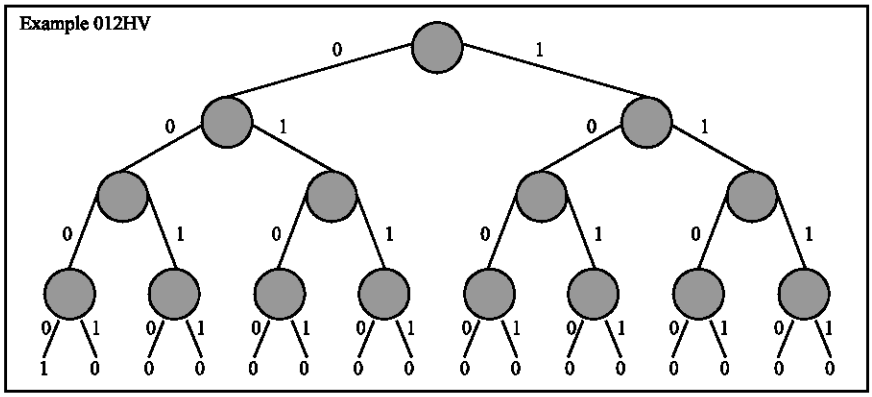
Algorithms (pseudo codes): Normalized polish expression has three constraints and use the Binary Decision Tree to implement two of three constraints.
The three constraints and their pseudo codes:
modules: operations (i.e., 1, 2, 3…)
vertical cuts or horizontal cuts: operators (i.e., V or H)

1. for every subexpression Ei = e1 … ei, 1 ≤ i ≤ 2n-1, # operands > # operators

(i.e., 1 2 H 3 4 V H)
operations: 1 2 2 3 4 4 4
operators: 0 0 1 1 1 2 3
Every node of the BDD decides a element of the polish expression to be a operator or a operation.
If (element I == operator)
{# operators++}
else{# operations++}
If (# operators < # operations)
{ the slicing tree is legal }
else{the slicing tree is illegal}



2. Skewed: no consecutive operators of the same type (H or V)
Every node of the BDD decides consecutive operators of the same or different type.
If (element I == element I-1)
{the slicing tree is non-skewed}
else{the slicing tree is skewed}[10,11]



3. The N operations need N-1 operators.
We can avoid this by adding a constraint at the input pattern generator. Complexity analysis
Every constraint only need complexity O(E). E is the length of input patterns, # elements.

FLOORPLANNING FLOWS

Because of the constraints of the CUDD, we create two flows to solve our problem.

Flow 1: optimum solutions

Flow 2: local optimum solutions

In the flow 1 we can get the optimum solutions successfully.

Step 1: Use the CUDD to build BDDs for constraint 1 and constraint 2.

Step 2: According to the result of the CUDD for constraint 1, we can get all polish expressions.

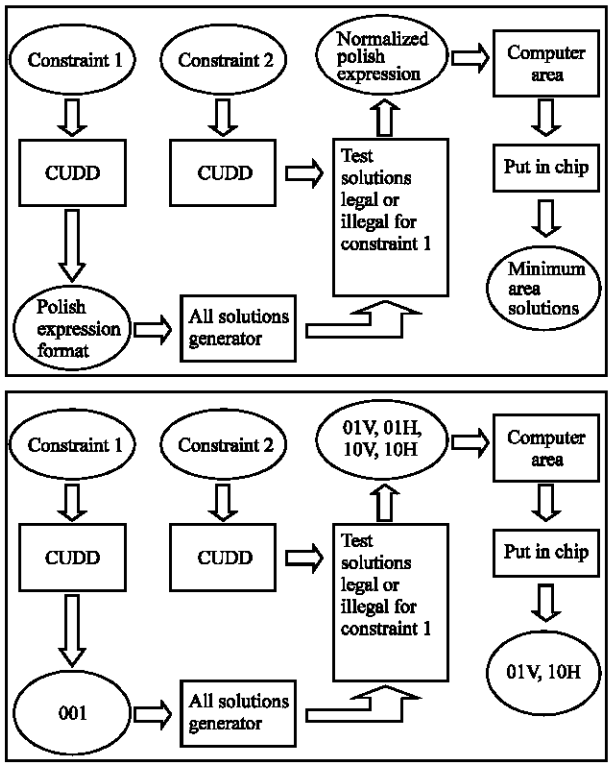
(i.e., 00011, 00101)
0: the place of operations
1: the place of operators

Step 3: According to the result of the CUDD for constraint 2 and all polish expressions, we can generator all normalized polish expressions.

(i.e., 00011, 00101)
012VH, 012HV, 01V2H, 01H2V...

Step 4: Compute the area of the all normalized polish expressions independently and choose the minimum one which can be putted into the chip area.

Step 5: The results of step 4 are the optimum solutions.



A. In the flow 2 we can get the usable solutions successfully.

Step 1: Use the random solutions generator to generator possible solutions.

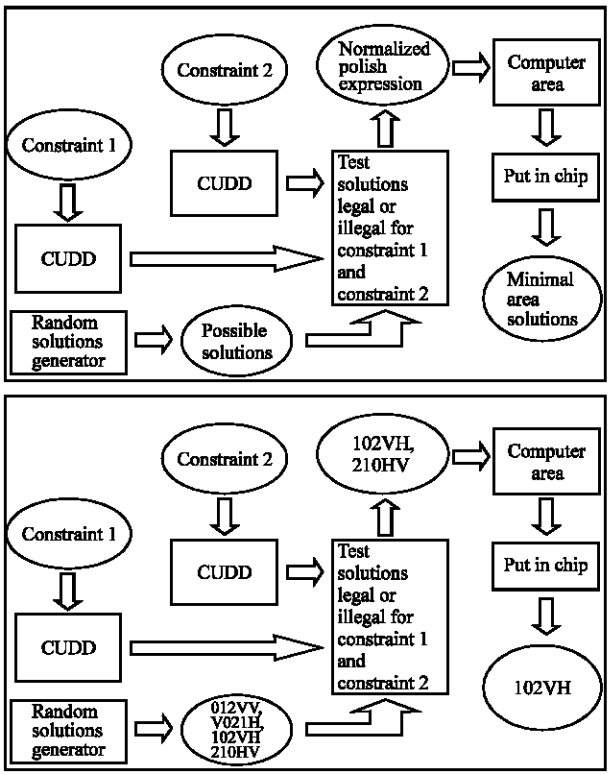
(i.e., 012VH, 0H12V, 012VV, 01V2H)

Step 2: According the results of the CUDD for constraint 1 and constraint 2, we can get some normalized polish expressions from the possible solutions.

(i.e., 012VH, 01V2H)

Step 3: Compute the area of the all normalized polish expressions from step 2 independently and choose the minimum one which can be putted into the chip area.

Step 4: The results of step 3 are the usable solutions and maybe the optimum solutions.



EXPERIMENTAL RESULTS

Flow 1: We define a big rectangle with chip length and chip width then we cut it into many small rectangles as showed in Fig. 4. If we can use flow 1 to put those small rectangles into the big rectangle, the flow 1 guarantees to get optimum solutions.

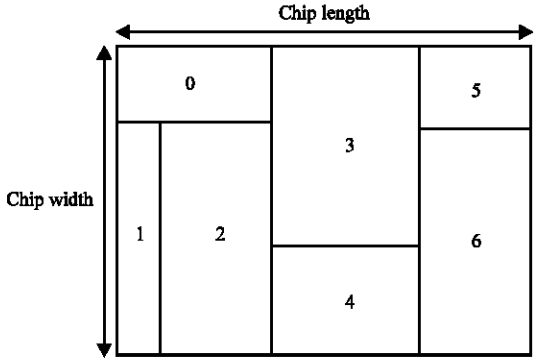


Fig. 4: Rectangle with chip length and width. Module 1,2,3,4,5,6

Table 1: Module number and Chip width/length in flow 1

Module number	2	3	4	5	6	7	8	9	10	11	12
Chip width	100	100	100	100	100	100	100	100			
Chip length	100	100	100	100	100	100	100	100			
Chip area	10000	10000	10000	10000	10000	10000	10000	10000			
Solution											
Min length	100	100	100	100	100						
Min width	100	100	100	100	100						
Min area	10000	10000	10000	10000	10000						
Some solutions	01V	012HV	301HV	301HV	3014H						
			2V	24HV	52HV						
					HV						
	01H	012VH	301V	301HV	3104H						
			2H	24HV	52HV						
					HV						
	10V	021HV	201HV	301HV	3014V						
			3V	24HV	52VH						
					VH						
Run time	0+15+	0+15+1	0+15+1	0+46+3	15+168	78+688	359+33	1578+	6906+	29968+	129234+
	0	6	25	156	7+1522	28	57359	∞	∞	∞	∞
					96						

Table 2: Module number and chip width/length in flow 2

Module number	2	3	4	5	6	7	8	9	10	11	12
Chip width	200	200	200	200	200	200	200	200	200	200	…
Chip length	200	200	200	200	200	200	200	200	200	200	…
Chip area	40000	40000	40000	40000	40000	40000	40000	40000	40000	40000	…
Solution											
Min length	100	100	100	120	140	130	200	115	140	115	…
Min width	100	100	100	100	80	90	65	120	95	105	…
Min area	10000	10000	10000	12000	11200	11700	13000	13800	13300	12075	…
Some solutions	01V	012HV	01V2H	30V1V	450H2	146V5	23V04	07V4H	521VH49VH	36V2V07V9	…
			3H	24HV	V31V	2VHV	V57V1	183VH	63H0H78VHV	4V15V10	…
					HV	0H3V	6HVH	256VH	V	8HVHVH	…
								V	VH		
	01H	021VH	32V10	30124							
			HV	VHVH							
	10V	21V0H	32H10	3012H							
			VH	V4VH							
Run time	16	31	16	15	15	31	62	78	62	78	…

Flow 1 is the flow we find the optimal solution with CUDD. At first, modeling the problem to which can be solved by CUDD. Then we get the BDD solution use those solutions to arrange all combination solution. Finally, we can compute the optimal solution in Table 1.

Flow 2: We define a bigger rectangle with double chip length and double chip width then random generator many possible solutions to put into the bigger rectangle (Fig. 5). Finally, we choose the minimum solutions to be the final solutions and the final solutions maybe the optimum solutions.

Flow 2 is the flow we find the optimal solution with CUDD. At first, modeling the problem to which we can be solved by CUDD. Then we get the BDD solution and use those solutions to randomly generate some solutions. Finally, we find the optimal solution among them in Table 2.

From Table 1 and 2, we can find that if we want to find the optimal solution. We will spend too much time

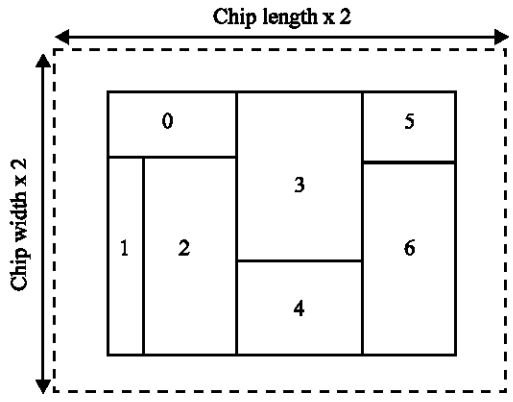


Fig. 5: Rectangle with double chip length and width. Module 0, 1, 2, 3, 4, 5, 6

computing all combination solutions. We just can handle up to 8 modules because of the limitation of running time and the huge solution output file (8 modules will output 12G Bytes file). But if we want a usable

solution, we can use flow2 to get some local optimal solutions. In flow 2, we can handle up to 11 modules because of limitation of memory space which is used to store the result of BDT. And the empty one means no-data.

CONCLUSIONS

In the flow 1 we can get the optimum solutions successfully, but the constraints, memory usage and running time, of the CUDD limit the number of modules which we can handle.

In the flow 2 we can get the local optimum solutions successfully, but the quality of solutions are depended on the random solutions.

REFERENCES

1. Puri, A. and T. Chen, 2001. Multimedia Systems, Standards and Networks, Marcel Dekke, pp: 211-214.
2. Sadka, A., 2002. Compressed Video Communications. John Wiley and Sons, pp: 211-214.
3. Richardson, I.E.G., 2002. Video Codec Design. John Wiley and Sons, pp: 179-183.
4. Yao-Wen, C., 2005. Physical Design handouts by Dr. Yao-Wen Chang (National Taiwan University, Taiwan), pp:11-25 .
5. James, H.K. and T.R. Shiple, 1998. Implicit State Enumeration for FSMs with Datapaths. In Proceedings of Formal Methods in Computer-Aided Design DAC '98 submission, pp: 122-125.
6. Monahan, C. and F. Brewer, 2003. Symbolic Modeling and Evaluation of Data Paths. Proc. 32nd ACMIEEE Design Automation Conf., pp: 313-317.
7. Shi-Yu, H. and K.T. Cheng, 2001. AQUILA: An equivalence checking system for large sequential designs. IEEE Trans. Computers, 49: 443-464.
8. Kuang-Chien, C., C.Y. Huang, F. Lu and C. Li, 2003. An efficient sequential SAT solver with improved search strategies. Proceeding Design Automation and Test Conference, pp: 1102-1107.
9. Ashar, P. and G.A. Devadas, 1991. Boolean satisfiability and equivalence checking using general binary decision diagrams. Proceeding of International Conference on Computer Design: VLSI in Computers and Processors, pp: 259 -264.
10. Mcgree, A., K. Truong, E.B. George, T.P. Barnwell and V. Viswanathan, 1996. A 2.4 kb/s MELP coder candidate for the new US federal standard. IEEEICASSP, 1: 200-203.
11. Park, S.H., Y.B. Kim, S.W. Kim and Y.S. Seo, 1997. Multi-Layer Bit-Sliced Bit-Rate Scalable Audio Coding. 103rd AES Convention, New York, Preprint 4520.