Optimization Methods for Lookup-Table-Based FPGAs Using Transduction Method

Shigeru Yamashita*,Yahiko Kambayashi* and Saburo Muroga**

*Integrated Media Environment Experimental Lab.,
Faculty of Engineering, Kyoto University.

**Department of Computer Science, University of Illinois.

Abstract— In recent years Field Programmable Gate Arrays(FPGAs) have emerged as an attractive means to implement low volume applications and prototypes due to their low cost, reprogrammability and rapid turnaround times. Therefore, the need for design methods of FPGAs are getting larger and larger. In this paper, two methods to optimize networks which have been mapped for lookup-table-based FPGAs are discussed. These methods utilize the notion of compatible sets of permissible functions(CSPFs) of Transduction Method. Experimental results show the effectiveness of our methods.

I. Introduction

In recent years, Field Programmable Gate Arrays(FPGAs) have emerged as an attractive means to implement low volume applications and prototypes due to their low cost, reprogrammability and rapid turnaround times. FPGAs also offer new possibility to design digital systems which can be easily reconfigured[2]. There are many types of commercially available FPGAs[2]. Lookup-table-based FPGAs are one of most popular types among them. Lookup-table-based FPGAs consist of logic blocks which can generate any functions of fixed numbers of input variables and programmable connections. In this paper, we focus on lookup-table-based FPGAs.

The traditional design flow for FPGAs consists of four steps[2]. In the first step, a logic optimizer performs technology independent optimization[1]. Then a technology mapper maps the circuit to logic blocks. Finally, placement and routing are done.

There exist a number of technology mappers for FPGAs, including: Chortle[4], mis-pga[7], Xmap[5], DAG-map[3]. These technology mappers map a Boolean network into a circuit of logic blocks. Since most conventional technology mappers divide networks into sets of logic blocks without considering the relationship between functions of logic blocks, there is a possibility to remain redundancy in mapped networks. In order to remove such redundancy, two methods, **Logic Block Substitution** and **Internal Logic Modification** are developed in this paper. These methods utilize the notion of *Compatible Sets of Permissible Functions* (CSPF) of Transduction Method[8]. The former method reduces number of logic blocks by substituting a logic block for another one. The latter method optimizes networks with modifying internal logics of logic blocks. This method is considered especially suitable for lookup-table-based FPGAs since performance of a chip is not affected only if internal logics of logic blocks are modified.

These methods were applied to the networks designed by MIS's technology mapper[7] for lookup-table-based FPGAs. It is shown that 8% reduction of logic blocks is obtained by the **Logic Block Substitution** on average. **Internal Logic Modification** attains 1% farther reduction.

The rest of this paper is organized as follows. In Section II, the networks we treat are defined and the basic terminology is explained. Two optimization methods for lookup-table based FPGAs are discussed in Section III. Experimental results of our methods are shown in Section IV. Our conclusion follows in Section V.

II. Basic Concepts and Definitions

In this section, the networks which we treat are defined and the basic terminology is explained. The main objective of this paper is to optimize the networks which have been mapped for lookup-table-based FPGAs. Therefore, loop-free multi-level combinational networks consisting of logic blocks and connections between them will be considered. Logic blocks can realize any function of fixed numbers of input variables.

A network can be viewed as a directed acyclic graph which consists of logic blocks as nodes and connections as edges. Let N be the number of logic blocks, n be the number of inputs of the network, $L = \{l_1, l_2, \cdots, l_N\}$ be the set of logic blocks and $C = \{c_{ij}\}$ be the set of connections where c_{ij} connects the output of l_i to an input of l_j. A logic block l_i is an immediate predecessor of l_j if there exists a connection c_{ij}. In that case l_j is an immediate successor of l_i.

$IP(l_i)$ and $IS(l_i)$ denote the set of all the immediate predecessors of l_i and the set of all the immediate successor of l_i, respectively.

Let $f(l_i)$ be the logic function realized at the logic block l_i. A function f at a logic block is represented with a 2^n-

dimensional vector, $f = (f^{(1)}, f^{(2)}, \cdots, f^{(2^n)})$ where $f^{(j)}$ is the value of f in the j-th row of the truth table for f. $f^{(j)}$ is 1, 0, or $*$, if the value of f in the j-th row of the truth table for f is 1, 0, don't care, respectively.

The **set of permissible functions**[8] of a logic block is the set of functions, where we can change the output function of the logic block to a member of them without changing the functionalities of the primary outputs of the network. There are two types of set of permissible functions, the maximum set of permissible functions (MSPF) and compatible sets of permissible functions (CSPF). MSPF of a logic block contains the largest set of permissible functions. On the other hand, CSPF of a logic block is a subset of its MSPF, where we can change functions of logic blocks to their CSPF at the same time. Let $G(l_i)$ be the CSPF of the logic block l_i.

III. Optimization Methods

In this section, two methods for reducing the number of logic blocks, **Logic Block Substitution** and **Internal Logic Modification** are shown.

A. Logic Block Substitution

As described below, **Logic Block Substitution** utilizes similar concepts of **Gate Substitution** of Transduction Method[8]. The procedure is formally stated as follows.

Logic Block Substitution

step1 Select a logic block as l_i one by one from the outputs toward the inputs of the network and go to step 2. If there is no logic block to select, halt.

step2 Select a logic block as l_j, such that l_j's level from the inputs is lower than l_i's and that $G(l_i)$ includes $f(l_j)$. If there is not such a logic block, go to step1. Otherwise, go to step3.

step3 If l_i is an output of the network, replace l_i by l_j, and go to step7. Otherwise, go to step4.

step4 Select a logic block as l_k from immediate successors of l_i one by one and go to step 5. If there is not such a logic block to select, go to step 7.

step5 If l_j is an immediate predecessor of l_k, go to step6. Otherwise, change the connection between l_i and l_k to the new connection between l_j and l_k, and go to step4.

step6 Disconnect l_i to l_k, change the internal logic of l_k properly and go to step 4.

step7 Delete l_i, along with the fanout free input cones of l_i. Go to step 1.

The reason why a lower level logic block l_j is selected at step2 is to avoid increase the number of levels of the network. Since l_j's level from the inputs is lower than l_i's, l_k cannot be a predecessor of l_j. Therefore, there is no possibility that the procedure makes a loop in a network at step5.

B. Internal Logic Modification

Logic Block Substitution discussed in Section Adoes not fully utilize the flexibility of logic blocks which can realize any logic functions with fixed numbers of fan-ins. Therefore, another optimization method called **Internal Logic Modification** has been developed. This method utilizes the flexibility of logic blocks.

Although **Logic Block Substitution** cannot replace l_i by l_j unless the output function of l_i is a member of the CSPF of l_j, **Internal Logic Modification** can replace l_i by l_j if the following conditions are satisfied.

- $G(l_i) \cap G(l_j)$ is not empty.
- $f(l_j)$ can be changed to $G(l_i) \cap G(l_j)$ only by modifying the internal logic of l_j.

The following procedure optimizes a network by modifying internal logics of logic blocks.

Internal Logic Modification

step1 Select a logic block as l_i one by one from the outputs toward the inputs of the network and go to step 2. If there is no logic block to select, halt.

step2 Select a logic block as l_j, such that l_j's level from the inputs is lower than l_i's and $G(l_i) \cap G(l_j)$ is not empty. If there is not such a logic block, go to step1. Otherwise, go to step3.

step3 Change $f(l_j)$ to $f'(l_j)$ which is included in $G(l_i) \cap G(l_j)$ by **SOP**(mentioned later). If **SOP** cannot do such transformation, go to step1. Otherwise, go to step4.

step4 If l_i is an output of the network, replace l_i by l_j, and go to step8. Otherwise, go to step4.

step5 Select a logic block as l_k from immediate successors of l_i one by one and go to step 6. If there is not such a logic block to select, go to step 8.

step6 If l_j is an immediate predecessor of l_k, go to step7. Otherwise, change the connection between l_i and l_k to the new connection between l_j and l_k, and go to step5.

step7 Disconnect l_i to l_k, change the internal logic of l_k properly and go to step 5.

step8 Delete l_i, along with the fanout free input cones of l_i. Go to step 1.

Internal Logic Modification is similar to **Logic Block Substitution** except step3. In the rest of this section, **SOP** is explained. The binary operation $\bullet$ is defined as Table I.

TABLE I
Binary Operator •

		Second element	
	•	0	1
First element	0	*	0
	1	*	1
	*	*	*

$SOP(F, f_1, f_2, \cdots, f_i)$ (SOP means Sum-Of-Products) returns a sum-of-products of $f_1, f_2, \cdots, f_i$ realizing F if possible, or **Error** if impossible, as shown in Figure 1. In Figure 1, **True** and **False** represent the function that is constantly 1 and 0, respectively. The operators '+' and '·' mean logical sum and logical product, respectively. These operators are defined to return **Error** if one of operands is **Error**.

```
$SOP(F, f_1, f_2, \cdots, f_i)$
    if(i = 1){
        if($f_1 \in F$) return $f_1$
        else if($\overline{f_1} \in F$) return $\overline{f_1}$
        else return Error
    }
    else{
        $F_1 = F \bullet f_1$
        if($False \in F_1$) $F_1$' = False
        else if($True \in F_1$) $F_1$' = True
        else {
            $F_1$' = $SOP(F_1, f_2, \cdots, f_i)$
            if($F_1$' = Error) return Error
        }
        $F_0 = F \bullet \overline{f_1}$
        if($False \in F_0$) $F_0$' = False
        else if($True \in F_0$) $F_0$' =True
        else {
            $F_0$' = $SOP(F_0, f_2, \cdots, f_i)$
            if($F_0$' = Error) return Error
        }
        return ($F_1$'·$f_1$ + $F_0$'·$\overline{f_1}$)
    }
```

Fig. 1. **SOP**

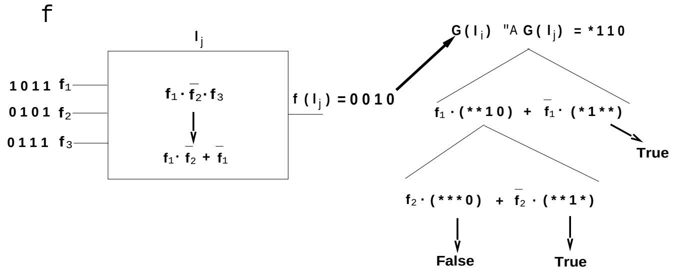
f
l_j
1011 f_1
0101 f_2
0111 f_3
$f_1 \cdot \overline{f_2} \cdot f_3$
$f_1 \cdot \overline{f_2} + \overline{f_1}$
$f(l_j)$ =0010
$G(l_i)$ "A $G(l_j)$ = *110
$f_1 \cdot (**10) + \overline{f_1} \cdot (*1**)$
True
$f_2 \cdot (***0) + \overline{f_2} \cdot (**1*)$
False
True

Fig. 2. Execution of **SOP**

For example, in the case of Figure 2 the procedure tries to change $f(l_j)$ to $(G(l_i) \cap G(l_j))$, i.e., $(*110)$. Let the logic functions of the inputs of l_i be f_1, f_2 and f_3 which are (1011), (0101) and (0111), respectively. Let the internal logic of l_j be expressed as $(f_1 \cdot \overline{f_2} \cdot f_3)$. **SOP** tries to expand $(*110)$ at f_1, f_2 and f_3 recursively. Thus, it expands $(*110)$ to $(F_1 \cdot f_1 + F_0 \cdot \overline{f_1})$ at first. F_1 is obtained by calculating $(*110) \bullet f_1$, i.e., $(**10)$. F_0 is obtained by calculating $(*110) \bullet \overline{f_1}$, i.e., $(*11*)$. Next, **SOP** expands F_1 at f_2 to obtain $(f_1 \cdot (***0) + F_0 \cdot \overline{f_1})$. It expands functions recursively until the following condition is satisfied.

- **SOP** obtains a logic function included in **True** or **False**. It means that no more expansion is needed.
- **SOP** has no more variable to expand. It means that **SOP** fails. In this case, **SOP** returns **Error**.

Finally, **SOP** successfully modifies the internal logic to $(f_1 \cdot \overline{f_2} + \overline{f_1})$ in order to change $f(l_j)$ to $G(l_i) \cap G(l_j)$ in this case.

IV. Implementation and Experimental Results

We have implemented the methods presented above. The SBDD package[6] was used to represent logic functions. MCNC benchmark circuits were used for experiments. In the experiments, a 5-input lookup-table architecture is assumed. The following commands of MIS's technology mapper[1] for lookup-table-based FPGAs were used to generate initial networks.

- xl_split -n 5
- xl_partition -n 5
- xl_cover

Logic Block Substitution and **Internal Logic Modification** are abbreviated as ”**BlockSub**” and ”**LogicModify**”, respectively.

The experimental results are shown in Table II. The third column in Table II shows the results of ”**BlockSub**”. The fourth column in Table II shows the results of ”**LogicModify**”.

From Table II, “**BlockSub**” reduces the numbers of logic blocks/connections by about 8% reduction on average. ”**BlockSub**”, however, can reduce levels of networks in only three cases. It is considered that there are too many critical paths in networks which have been mapped for lookup-table-based FPGAs.

From Table II, the numbers of logic blocks of ”**LogicModify**” are less than that of ”**BlockSub**” by about 1% on average. “**LogicModify**”, however, does not always show better results than “**BlockSub**”. The reason is considered as follows. ”**LogicModify**” can replace more logic blocks than ”**BlockSub**”, since ”**LogicModify**” can modify internal logics of logic blocks if necessary. However, if logic blocks to replace are not selected in good order, there are such cases that the final result may be worse than that of ”**BlockSub**” as the following example. A logic block l_i which “**BlockSub**” cannot deal with

TABLE II
Experimental Results

Circuits	Initial			BlockSub				LogicModify			
	LB	conn	lev	LB	conn	lev	CPU	LB	conn	lev	CPU
C432	122	317	13	109	282	13	25.7	109	**277**	13	68.0
C499	243	408	11	**211**	376	11	621.3	217	**370**	11	821.4
alu2	141	540	22	138	533	22	4.7	**137**	**526**	22	4.3
alu4	264	891	25	260	879	24	19.4	**258**	**870**	24	18.5
apex7	129	318	7	128	313	7	2.7	128	313	7	4.0
b9	90	197	3	88	191	3	2.1	**87**	**188**	3	2.7
c8	87	235	4	**84**	**224**	4	1.8	85	228	4	2.2
cordic	55	107	7	**50**	**94**	7	1.8	51	96	6	2.1
example2	195	445	5	**189**	**422**	4	2.4	192	433	4	5.5
i9	478	1490	9	443	1407	**7**	50.2	**420**	**1262**	9	253.3
lal	83	213	5	**73**	**179**	5	1.5	75	183	5	1.8
sct	65	169	4	**61**	**160**	4	1.4	62	162	4	1.8
term1	173	553	8	149	460	8	11.8	**141**	**424**	8	21.3
too_large	352	1250	13	295	1004	13	294.5	**280**	**960**	**11**	7821.3
vda	400	1683	5	**355**	**1458**	5	6.8	367	1518	5	11.9

LB: numbers of logic blocks
conn: numbers of connections
lev: numbers of levels of the networks
CPU: CPU time running on a SPARC 10 (sec.)
Bold numbers show better cases.

is replaced by **"LogicModify"**. The configuration of the network, however, becomes worse for the later transformation by replacing the logic block l_i. In this case, final result may become worse by replacing the logic block l_i. In the implemented method, order of logic blocks to apply **"LogicModify"** was not considered. Finding the best order of logic blocks to apply **"LogicModify"** is a future work. We expect that better results can be obtained if proper order is found.

Although the initial circuits used in our experiments are not the best possible ones due to availability, we will try to obtain other circuits for further experiments.

V. Conclusions

In this paper, the notion of Transduction Method was applied to lookup-table-based FPGAs. Two optimization methods for lookup-table-based FPGAs, **Logic Block Substitution** and **Internal Logic Modification**, were presented. Experimental results of our methods were also presented. The experimental results show that our methods reduce the numbers of logic blocks by 8 % on average from initial circuits designed by MIS's technology mapper. We plan to develop another optimization methods especially for level reduction and to combine with placement and routing phases.

Acknowledgements

We would thank Shuzo Yajima for permitting to use the SBDD package developed by his group. This research was supported by International Research Program : Joint Research from the Ministry of Education, Science and Culture in Japan.

References

[1] R. Brayton, E. Detjens, S. Krishna, T.Ma, P. McGeer, L.Pei, N. Phillips, R. Rudell, R. Seagal, A. Wang, R. Yung, A. Sangiovanni-Vincentelli," Multiple-level logic optimization system", *Proceedings of International Conference on Computer-Aided Design*, (Nov. 1986), 356-359.

[2] S.D.Brown, R.J.Francis, J.Rose, Z.G.Vranesic, "Field-programmable gate arrays", (Kluwer Academic Publishers, 1992).

[3] K.C.Chen, J.Cong, Y.Ding, A.B.Kahng, P.Trajmar,"DAG-map:Graph-based FPGA technology mapping for delay optimization", *IEEE Design and Test of Computers*, (Sept. 1992), 7-20.

[4] R.J.Francis, J.Rose, K. Chung: Chortle," A techonology mapping program for lookup table-based FPGAs", *Proceedings of 27th Design Automation Conference*, (June 1990), 613-619.

[5] K.Karplus," Xmap: A technology mapper for table-lookup FPGAs", *Proceedings of 28th Design Automation Conference*, (June 1991), 240-243.

[6] S. Minato, N. Ishiura, S. Yajima: Shared binary decision diagram with attributed edges for efficient boolean function manipulation, *Proceedings of 27th Design Automation Conference*,(June 1990),52-57.

[7] R. Murgai, Y. Nishizaki, N. Shenoy, R. Brayton, A. Sangiovanni-Vinccentelli: Logic Synthesis for Programmable Gate Arrays, *Proceedings of 27th Design Automation Conference*, (June 1990), 620-625.

[8] S. Muroga, Y. Kambayashi, H. C. Lai, J. N. Culliney: The Transduction Method-Design of Logic Networks Based on Permissible Functions, *IEEE Transactions on Computers*, 38-10(Oct. 1989), 1404-1424.