Evolutionary Synthesis of Multiplexor Circuits under Hardware Constraints

Rolf Drechsler **Wolfgang Günther**

Institute of Computer Science, Chair of Computer Architecture (Prof. Dr. Bernd Becker)
Am Flughafen 17, Albert-Ludwigs-University, 79110 Freiburg im Breisgau, Germany
email: {drechsle,guenther}@informatik.uni-freiburg.de

Abstract

Multiplexor based Field Programmable Gate Arrays (FPGAs) are a very popular design style. However, during circuit design only limited hardware resources are available. In this paper we present an approach to multiplexor (MUX) based circuit evolution under hardware constraints. Binary Decision Diagrams (BDDs) are used as the underlying data structure. An evolutionary technique is proposed that allows to stay within given hardware resources while minimizing the error in parallel. Starting from a correct circuit, i.e. a BDD representing the complete function, some parts are removed, based on evolutionary principles, until all constraints are met. The experimental results presented demonstrate the possibility to handle for the first time instances composed of several thousand gates.

1 Introduction

Field Programmable Gate Arrays (FPGAs) have been widely used in implementations of integrated circuits due to several reasons, like short turnaround time and time-to-market aspects. FPGAs are mainly based on a basic cell that consists e.g. of a *Look-Up Table* (LUT) or a *Multiplexor* (MUX) structure (Brown *et al.*, 1992; Murgai *et al.*, 1995).

Binary Decision Diagrams (BDDs) are the state-of-the-art data structure in VLSI CAD and have been used in many applications (Drechsler & Becker, 1998). In the meantime they have also been integrated in industrial tools. BDDs are graph based representations of Boolean functions where in each node a Shannon decomposition is carried out. Furthermore, many operations can be carried out efficiently, like satisfiability and Boolean AND. BDDs can easily be mapped to MUX based FPGAs due to the close relation between BDD nodes and MUXs. Several logic synthesis approaches based on BDDs have been developed (Buch *et al.*, 1997; Chaudhry *et al.*, 1998; Günther & Drechsler, 1999).

Also in the field of *Evolvable Hardware* (EHW) BDDs have gained large attention, since they are easy to manipulate (Droste, 1997). They are also well-suited for theoretical analysis (Droste *et al.*, 1999). One main problem in circuit design is the limited availability of hardware resources. This especially becomes a problem in EHW where the circuits tend to grow over time for realizing a given target function.

In this paper we present a method for circuit evolution under hardware constraints based on BDDs. The resulting netlists are mapped to MUX based FPGAs. In contrast to previously published approaches (see e.g. (Kalganova & Miller, 1999)) we start with a complete description of the function given as a BDD. Assuming that the complete graph does not fit on the FPGA we set an upper limit on the number of MUX cells. This means that the BDD has to be modified to fit on the FPGA. We propose an algorithm based on evolutionary techniques that minimizes the error introduced by this modification. Nodes from the BDD are removed until the hardware requirements are met. Due to this technique the algorithm runs very fast, i.e. for the first time functions with more than 200 variables can be handled, while other approaches fail for functions with more than 10 variables (Thompson, 1997; Kalganova & Miller, 1999).

Experimental results are given that demonstrate the trade-off between area of the circuit and error introduced. It is shown that our algorithm results in an error of less than 1% on average, if the area is reduced by 20%.

2 Preliminaries

2.1 Binary Decision Diagrams

Boolean variables can assume values from $\mathbf{B} := \{0, 1\}$. In the following, we consider Boolean functions $f : \mathbf{B}^n \rightarrow \mathbf{B}^m$ over the variable set $X_n = \{x_1, \ldots, x_n\}$.

It is well-known that each Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ can be represented by a *Binary Decision Diagram* (BDD) i.e. a directed acyclic graph where a Shannon decomposition

$$f = \overline{x}_i f_{x_i=0} + x_i f_{x_i=1} \quad (1 \leq i \leq n)$$

is carried out in each node.

A BDD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal node and if the variables are encountered in the same order on all such paths. A BDD is called *reduced* if it contains neither isomorphic sub-graphs nor vertices with both edges pointing to the same node.

BDDs are defined analogously for multi-output functions $f : \mathbf{B}^n \rightarrow \mathbf{B}^m$ as for the case of single-output functions: A BDD G_j for each component function f_j $(1 \leq j \leq m)$ is used for the *shared* BDD representation G for f. The order of the variables is fixed over all G_js.

For functions represented by reduced, ordered BDDs efficient manipulations are possible (Bryant, 1986). In the following, only reduced, ordered BDDs are considered and for briefness these graphs are called BDDs.

2.2 Multiplexor Circuits

In general, a *Combinational Logic Circuit* (CLC) is defined over a fixed library and modeled as a directed acyclic graph $C = (V, E)$ with some additional properties: each vertex $v \in V$ is labeled with the name of a basic cell or with the name of a *Primary Input* (PI) or *Primary Output* (PO). The collection of basic cells available is given by a fixed library. Of course, basic cells with arbitrary complexity, especially with an arbitrary number of inputs and outputs, are possible. There is an edge (u, v) in E from vertex u to v, iff an output pin of the cell associated to u is connected to an input pin of the cell associated to v, i.e., edges contain additional information to specify the pins of the source and sink node they are connected to. Vertices have exactly one incoming edge per input pin. Nodes labeled as PI (PO) have no incoming (outcoming) edges.

Very often a standard library (*STD*) consisting of the 2-input, 1-output *AND, OR* gate and the 1-input, 1-output inverter *NOT* is used. The circuits, as they

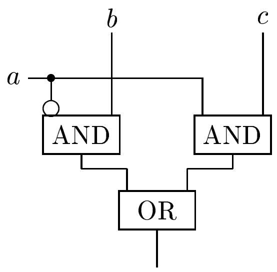


Figure 1: Multiplexer cell *MUX*

are defined in this paper, are CLCs over the library *MUXLIB*. *MUXLIB* consists of the library *STD* supplemented by a *multiplexer cell MUX*. We use the standard AND-, OR-, INVERTER-based realization of a *MUX* as it is shown in Figure 1. The input a is called *control input*, the input b (c) is called *0-input (1-input)*. The 0-input and the 1-input are called data inputs of the multiplexer.

A multiplexer-circuit C_1 over *MUXLIB* can easily be interpreted as a circuit C_2 over *STD* just by replacing each basic cell by its standard cell realization. C_2 is then called an *expansion* of C_1.

2.3 Relation between BDDs and MUX Circuits

It is well-known, that BDDs directly correspond to multiplexer based circuits called *BDD-circuits* in this paper. They are defined over the library *MUXLIB*.

A BDD-circuit can easily be obtained from a BDD by traversing the BDD in topological order and replacing all nodes by the corresponding multiplexers from the library *MUXLIB*. For a more detailed description of the algorithm see (Becker, 1992).

In the following we present an example from (Becker, 1992), which shows the transformation from the initial BDD to a circuit.

Example 1 Let $f(x_1, .., x_4) = x_1x_3 + x_2x_4 + \overline{x}_3x_4$. The BDD for f is shown in Figure 2, where each label i in a node corresponds to the variable x_i. The resulting circuit is shown in Figure 3.

3 Problem Formulation

On FPGAs only a limited number of cells is available. For this, in EHW approaches it happens that the complete function does not fit on the chip. In these cases a larger device has to be chosen, what usually results

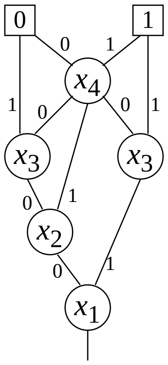


Figure 2: BDD for function f

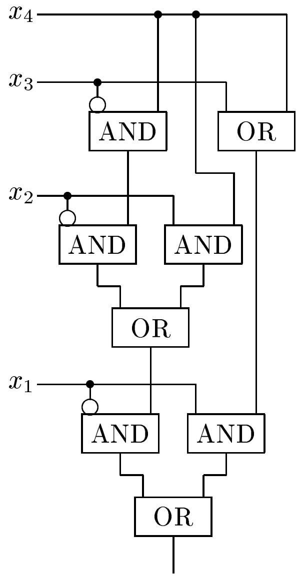


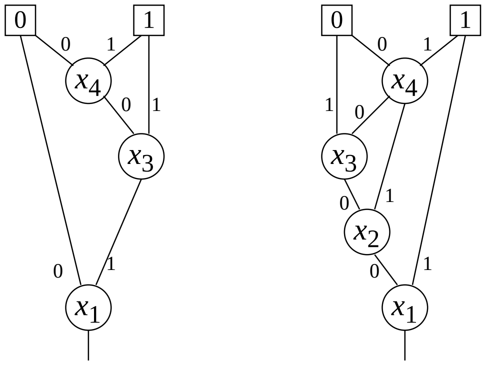
Figure 3: Circuit for function f

in higher costs, or the function to be realized has to be modified. In the second case the interest is to keep the amount of changes as small as possible.

We now consider the following problem that will be solved using evolutionary synthesis:

> *How can the number of nodes in a BDD be reduced (by a given factor) such that the resulting function is as similar to the original one as possible?*

Here similarity is measured in the number of common minterm values, i.e. the number of assignments for which both functions have the same value. To avoid huge numbers, the number of minterms is normalized to be in the range of $[0.0, 1.0]$ by dividing the minterm



(a) (b)

Figure 4: Substituting nodes with constants

count by 2^n. Nevertheless we will refer to this relative number of minterms as "the minterms" in the following.

Example 2 Consider again the function from Example 1. The minterms of f can be computed using the minterms of the two successor nodes by

$$\begin{aligned} minterms(f) = \tfrac{1}{2}(&minterms(f_{x_i=0}) \\ &+minterms(f_{x_i=1})). \end{aligned}$$

For the nodes of the BDD from Figure 2 the minterms are given by

const 0	0
const 1	1
x_4	1/2
x_3 (left)	1/4
x_3 (right)	3/4
x_2	3/8
x_1	9/16

To reduce the size of the BDD, we substitute nodes by constants. For example, substituting the node labeled by x_2 by the constant 0 results in the BDD given in Figure 4 (a). The resulting function has a minterm count of 6/16; 3/16 of the minterms are different from the original function. However, if the right node labeled with x_3 is substituted by the constant 1, then the resulting number of minterms is 11/16 (see Figure 4 (b)), and only 2/16 of the minterms are different.

Remark 1 The difference of two functions is computed by building the EXOR of both BDDs and counting the number of satisfying assignments of the resulting BDD. This computation can be done efficiently on BDDs. For multi-output functions, the maximum error of each pair of outputs is used.

It can be seen that selecting nodes has a large impact on the quality of the resulting BDD. In the following, methods are presented that are based on this observation.

4 Evolutionary Synthesis

4.1 Random Approach

A straightforward approach is to randomly select a node of the BDD and replace it by a constant. This process can be iterated until the maximum size that is allowed is reached. However, the quality of this approach is not satisfying in most cases. Therefore, in the following two more sophisticated methods are presented. The first one is a greedy approach, while the second one is based on evolutionary techniques.

4.2 Minterm Approach

Instead of selecting a random node of the BDD, a "good" node is chosen: if a node has only very few minterms, then it can be replaced by the constant zero without introducing much error. If, on the other hand, its onset nearly covers the complete space, then this node can be replaced by the constant one. This motivates the following approach, as given in Figure 5: First, the minterm count is computed for each node of the BDD (this is possible in linear time in terms of the number of nodes (Bryant, 1986)). Then the node having the most extreme minterm number (i.e. close to 0.0 or close to 1.0) is selected and replaced by the according constant. This is repeated as long as the BDD size exceeds the given limit.

```
minterm_algorithm(BDD f, int maximum_size) {
    while (size(f) > maximum_size) {
        compute_minterm_of_each_node(f);
        min = ∞;
        foreach node g of f {
            if (minterms(g) ≤ 0.5 and
                    minterms(g) < min) {
                min = minterms(g);
                g_min = g;
            }
            if (minterms(g) ≥ 0.5 and
                    1−minterms(g) < min) {
                min = 1− minterms(g);
                g_min = g;
            }
        }
        if (min ≤ 0.5)
            replace_node(f, g, 0);
        else replace_node(f, g, 1);
    }
    return f;
}
```

Figure 5: Minterm approach

4.3 Evolutionary Approach

To further improve the quality of the results, an evolutionary approach is presented in the following. It is based on one crossover operator and two different mutation operators:

MUT1: One randomly chosen node is replaced by a constant value (in the same way as done in the straightforward approach).

MUT2: The best node (in terms of minterm count, see Section 4.2) is replaced by a constant value.

CROSSOVER: The algorithm of Figure 6 is used to combine two functions f and g. Basically, it combines substitutions that are made by either of the two given functions. The algorithm recursively traverses the BDDs for f and g. If either of the operands is constant, then a constant value is returned. Otherwise the left and right sons are computed recursively and a node representing both sub-functions is returned. Intermediate results are stored in a computed table to ensure that the algorithm has linear runtime (in terms of the graph size). Note that in case the functions are equal, no modification takes place.

The initial population is generated using the original BDD as first individuum and by applying mutation operator MUT1 to create further elements. Evaluation is done using a combination of the BDD size and the percentage of wrong minterms compared to the original function, i.e. the following code is used:

```
evaluate(element e) {
    cost = error_weight · error(e, original_func);
    size = BDD_size(e) − maximum_size;
    if (size > 0)
        total_cost = cost + size;
    else total_cost = cost;
    return total_cost;
}
```

Initially, *error_weight* is set to 65536. It is divided by 2 if no improvement has been observed for the last 100 iterations while the BDD size of the best element is still too large. A sketch of the overall algorithm is given in Figure 7.

```
evolutionary_algorithm(BDD f, int maximum_size) {
    generate_initial_population();
    do {
        for (each child i) {
            j = linear_ranking_selection();
            randomly_select_method:
            case MUT1: child(i) = MUT1(element j);
            case MUT2: child(i) = MUT2(element j);
            case CROSSOVER: k = linear_ranking_selection();
                            child(i) = CROSS(element j and k);
        }
        update_population();
        if (age_of_best_element ≥ 100 and size_of_best_element > maximum_size) {
            error_weight = error_weight / 2;
            evaluate all elements again;
            set age of best element to 0;
    } until (age_of_best_element ≥ 100 and size_of_best_element ≤ maximum_size);
    return best element;
}
```

Figure 7: Sketch of evolutionary algorithm

```
crossover(BDD f, BDD g) {
    if (result is in computed table)
        return result;
    if (f is constant) {
        if (g is constant)
            return either f or g (at random);
        else
            return f;
    }
    if (g is constant)
        return g; /* f is non-constant here */
    /* now f and g are non-constant */
    x = top variable of f and g;
    t = crossover(f_{x=1}, g_{x=1});
    e = crossover(f_{x=0}, g_{x=0});
    return node(x, t, e);
}
```

Figure 6: Crossover operator

5 Experimental Results

In this section we describe our experimental results. All experiments are carried out on a *SUN Ultra 4* workstation. The CUDD package (Somenzi, 1998) is used as underlying BDD package.

We consider functions from (Brglez & Fujiwara, 1985). Some statistics are given in Table 1 to give an impression on the complexity of the benchmarks considered. The name of the circuit is given in column *name*. The number of inputs (outputs) is shown in column *in* (*out*). The size of the function in number of gates and number of BDD nodes is given in the last two columns, respectively. As can be seen, the benchmarks have up to 250 inputs and more than 100 outputs and may contain several thousand gates. Due to the sizes of the instances considered high demands on the efficiency of the algorithms are set. The runtime of our evolutionary algorithm described in the previous section varies from 100 to 14000 CPU seconds.

Table 1: Statistics of the benchmark set

name	in	out	gates	BDD nodes
c1355	41	32	514	29561
c1908	33	25	880	6252
c2670	233	140	1161	3980
c3540	50	22	1667	23828
c432	36	7	160	1209
c499	41	32	202	26407
c5315	178	123	2290	1777
c7552	207	108	3466	8407
c880	60	26	357	8411
i10	257	224	2497	52616

In Table 2 the results of our approach is given. Again, in the first column the name of the benchmark is shown. The next two columns (*rand*) show the resulting error in percent, if 50% and 20% of the BDD nodes

Table 2: Experimental results

name	rand 50%	rand 20%	greedy 50%	greedy 20%	evolutionary algorithm 50%	evolutionary algorithm 20%
c1355	50.0000	50.0000	**0.3906**	**0.3906**	1.0281	0.4688
c1908	50.0000	50.0000	**0.1587**	0.1587	0.2476	**0.1476**
c2670	37.5000	25.0000	3.3073	**0.0004**	**0.9552**	**0.0004**
c3540	50.0000	23.0774	17.4372	**0.9691**	**3.5583**	1.4491
c432	50.9952	21.7765	29.3719	13.0580	**12.0272**	**3.7693**
c499	50.0000	50.0000	**0.3906**	**0.3906**	0.4730	**0.3906**
c5315	50.0000	50.0000	37.5000	2.0428	**7.0312**	**0.2197**
c7552	50.0000	50.0000	50.0000	6.2500	**11.9003**	**0.7706**
c880	50.0000	26.2762	12.5000	6.2500	**1.0373**	**0.3890**
i10	50.0000	50.0000	3.1774	1.9039	**1.7584**	**0.8685**

are randomly removed, respectively (see Section 4.1). The columns *minterm* give the same information, if instead of random deletion a clever greedy algorithm based on the number of minterms is used (see Section 4.2). The results for our evolutionary synthesis approach are given in the last two columns. The best numbers are given in bold.

As can be seen, the evolutionary approach clearly outperforms not only random deletion, but also the minterm based algorithm. For the first time, instances of several thousand gates can be handled. If 20% of the nodes are removed, in most cases the error is less than 1%. In some cases the error is less than 10^{-5} (see *c2670*). Even if 50% are deleted, the error is below 10% in many cases using the evolutionary algorithm, while it often exceeds this limit using the minterm approach.

6 Conclusion

We presented an approach to hardware design based on evolutionary synthesis. The technique considers hardware constraints and trades off area versus erroneous behavior. Experimental results have shown that in most cases the area can be significantly reduced, i.e. by more than 20%, while the error never exceeds 4%.

References

Becker, B. 1992. Synthesis for Testability: Binary Decision Diagrams. *Pages 501–512 of: STACS*. LNCS, vol. 577. Springer Verlag.

Brglez, F., & Fujiwara, H. 1985. A Neutral Netlist of 10 Combinational Circuits and a Target Translator in Fortran. *Pages 663–698 of: Int'l Symp. Circ. and Systems, Special Sess. on ATPG and Fault Simulation.*

Brown, S.D., Francis, R.J., Rose, J., & Vranesic, Z.G. 1992. *Field-Programmable Gate Arrays*. Kluwer Academic Publisher.

Bryant, R.E. 1986. Graph - Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comp.*, **35**(8), 677–691.

Buch, P., Narayan, A., Newton, A.R., & Sangiovanni-Vincentelli, A.L. 1997. Logic Synthesis for Large Pass Transistor Circuits. *Pages 663–670 of: Int'l Conf. on CAD.*

Chaudhry, R., Liu, T.-H., Aziz, A., & Burns, J.L. 1998. Area-Oriented Synthesis for Pass-Transistor Logic. *Pages 160–167 of: Int'l Conf. on Comp. Design.*

Drechsler, R., & Becker, B. 1998. *Binary Decision Diagrams – Theory and Implementation*. Kluwer Academic Publishers.

Droste, S. 1997. Efficient genetic programming for finding good generalizing Boolean functions. *Pages 82–87 of: Genetic Programming Conference.*

Droste, S., Janssen, T., & Wegener, I. 1999. Perhaps Not a Free Lunch But At Least a Free Appetizer. *Pages 833–839 of: Genetic and Evolutionary Computation Conference.*

Günther, W., & Drechsler, R. 1999. ACTion: Combining Technology Mapping and Logic Synthesis for MUX based FPGAs. *Pages 125–132 of: E.I.S.-Workshop.*

Kalganova, T., & Miller, J. 1999. Evolving more efficient digital circuits by allowing circuit layout evolution and multiobjective fitness. *Pages 54–63 of: NASA/DoD Workshop on Evolvable Hardware.*

Murgai, R., Brayton, R.K., & Sangiovanni-Vincentelli, A.L. 1995. *Logic Synthesis for Field-Programmable Gate Arrays*. Kluwer Academic Publisher.

Somenzi, F. 1998. *CUDD: CU Decision Diagram Package Release 2.3.0.* University of Colorado at Boulder.

Thompson, A. 1997. *Hardware Evolution: Automativ Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution.* Springer Verlag.