

# DDBDD: Delay-Driven BDD Synthesis for FPGAs

Lei Cheng,  
 CS Dept & Coordinated Science Lab  
 University of Illinois at Urbana-Champaign  
 lcheng1@uiuc.edu

Deming Chen, Martin D.F. Wong  
 ECE Dept & Coordinated Science Lab  
 University of Illinois at Urbana-Champaign  
 {dchen,mdfwong}@uiuc.edu

## ABSTRACT

In this paper, we target FPGA performance optimization using a novel BDD (binary decision graph)-based synthesis approach. Most of previous works have focused on BDD size reduction during logic synthesis. In this work, we concentrate on delay reduction and conclude that there is a large optimization margin through BDD synthesis for FPGA performance optimization. Our contributions are threefold: (1) we propose a gain-based clustering and partial collapsing algorithm to prepare the initial design for BDD synthesis for better delay; (2) we use a technique named linear expansion for BDD decomposition, which in turn enables a dynamic programming algorithm to efficiently search through the optimization space for the BDD of each node in the clustered circuit; (3) we consider special decomposition scenarios coupled with linear expansion for further improvement on quality of results. Experimental results show that we can achieve a 95% gain in terms of network depths, and a 20% gain in terms of routed delay, with a 22% area overhead on average compared to a previous state-of-art BDD-based FPGA synthesis tool, BDS-pga.

## Categories and Subject Descriptors

J.6 [Computer-Aided Engineering]: Computer Aided Design

## General Terms

Algorithm, Performance

## Keywords

FPGA technology mapping, binary decision diagrams, linear expansion

## 1. INTRODUCTION

In LUT-based FPGA architecture, the basic programmable logic element is a K-input lookup table. A K-input LUT (K-LUT) can implement any Boolean functions of up to K variables. The conventional FPGA logic synthesis flow starts with a logic optimization phase, which is followed by a gate decomposition phase, and then a technology mapping phase. During the course of logic optimization, each node of the network can be simplified using a two-level logic optimizer, such as ESPRESSO [1], based on the *don't cares* extracted from the network or provided by the user [2,3]. After logic optimization, gate decomposition algorithms, such as the *tech\_decomp* in SIS [4] and the *dmig* in [5], are always carried out to decompose large-fanin gates into small-fanin gates so that every gate of the network is with a fanin number  $\leq K$ , where  $K$  is the input size of the LUT in the target FPGA. Then a technology mapping [6–8] algorithm is used

to convert the circuit into a functionally equivalent network comprised only of logic cells implementable in LUTs. The design is finished by placing these cells on an FPGA chip, and programming the connections among them.

While the traditional logic optimization methodology is very successful on AND/OR-intensive circuits, its performance on XOR-intensive circuits is far from satisfactory [9]. In [9], the authors presented BDS, a logic optimization system based on BDD decomposition techniques. By exploring the structure of a binary decision diagram, BDS is able to identify not only AND/OR decompositions, but also XOR/MUX decompositions. BDS has been successfully applied to FPGA designs in [10]. The logic synthesis system presented in [10], BDS-pga, first collapses the network using a maximum fanout free cone (MFFC)-based eliminating method, then it cuts BDDs in the middle recursively for LUT decomposition. After each cut, it tries to further reduce the number of mapped LUTs by swapping variable orders in the BDD. BDS-pga has shown significant improvements on both area and delay for some circuits [10] compared to SIS+Flowmap [7]. Another BDD-based synthesis technique is introduced in [11], where BDD re-synthesis is applied to improve timing. After placement, some timing critical parts of the circuit are selected, re-synthesized, and then re-placed.

One drawback of BDS, as mentioned in [9], is its inability of considering delay optimization, because it can not properly balance the factoring tree used in their algorithm. To optimize the delay, BDS-pga uses a delay re-synthesis approach. After logic synthesis, BDS-pga finds out critical paths and partially collapses these paths. Then, it uses ESPRESSO [1] algorithm to optimize the collapsed nodes, and re-decomposes the optimized nodes for delay optimization. This method is similar to what SIS [4] does for delay optimization. However, because the delay optimization is not integrated within the main logic synthesis algorithm, it does not always perform well as shown in our experiments.

In this paper, we propose a BDD-based FPGA logic synthesis system targeting delay optimization (under unit delay model). We introduce a gain-based partial collapsing algorithm considering delay. We use BDDs to represent node functions, and use linear expansion for BDD decomposition - a generalized decomposition technique - to synthesize the circuit. Based on linear expansion, we design a dynamic programming algorithm to choose the proper decompositions of a BDD to optimize delay. We also consider special decomposition scenarios that can be coupled with linear expansion for further improvement on quality of results. Experimental results show that we can achieve up to 90% gain in terms of mapped depth with 22% area overhead compared to BDS-pga.

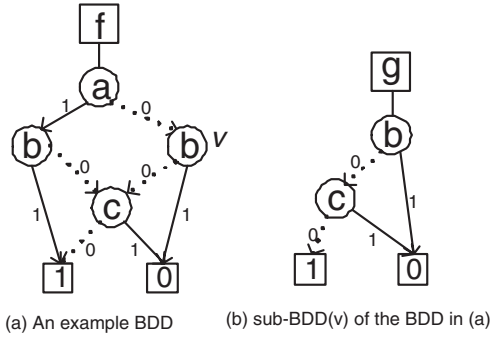
In Section 2, we introduce some related terminologies and preliminaries. Section 3 presents our algorithm in detail. Experimental results are shown in Section 4, and we conclude this paper in Section 5.

## 2. DEFINITIONS AND PRELIMINARIES

We assume that the readers are familiar with basic concepts of Boolean functions, Boolean networks, and BDDs [12]. We provide a brief review of related concepts, and define several terminologies used in the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2007, June 4–8, 2007, San Diego, California, USA.  
 Copyright 2007 ACM 978-1-59593-627-1/07/0006 ...\$5.00.



**Figure 1:** (a) BDD for the Boolean function :  $f = a.b.v.b.\bar{c}$ , and the variable inside each node is the input variable associated with the node; (b) sub-BDD( $v$ ) of the BDD in (a).

## 2.1 Boolean Functions and BDDs

A completely specified Boolean function with  $n$ -inputs and one output is a mapping  $f : B^n \rightarrow B$ , where  $B = \{0, 1\}$ . The *support* of Boolean function  $f$ , denoted  $supp(f)$ , is the set of variables on which  $f$  explicitly depends. A *Boolean network* is a directed acyclic graph (DAG), whose nodes represent Boolean functions. In this paper, the term *Boolean function* is used for a completely specified Boolean function.

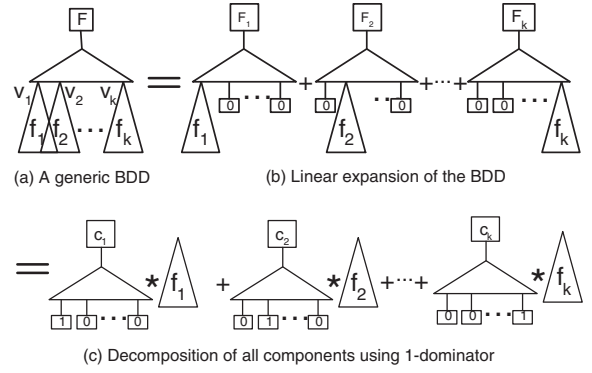
Binary decision diagrams (BDDs) were first introduced by Lee [13], and then popularized by AKers [14]. In [12], Bryant introduced the concept of reduced ordered BDDs (ROBDDs) and a set of efficient operators for their manipulation, and proved the canonicity property of ROBDDs. A binary decision diagram is a DAG, representing a Boolean function, with two terminal nodes (terminal node 1 and terminal node 0). Each non-terminal node has an index to identify an input variable of the Boolean function and has two outgoing edges, called the 0-edge and 1-edge. In the BDD drawings of this paper, we use solid lines to represent 1-edges, and dotted lines to represent 0-edges. The depth (or level) of a BDD is the number of input variables of the BDD. An ROBDD is a BDD where input variables appear in a fixed order in all the paths of the graph and no variables appear twice in a path, and every node represents a distinct function. In this paper, we refer ROBDDs as BDDs. Fig. 1(a) is a BDD example<sup>1</sup>. The size of a BDD can be reduced by *complement edges*, which point to the complementary form of the functions. To maintain canonicity, a complement edge can only be assigned to the 0-edge [15].

For simplicity, it is assumed that all the discussions in this paper are within the context of a BDD. The *root* of a BDD is the node without any incoming 0-edge or 1-edge, such as node  $a$  in Fig. 1(a). Let  $\mathcal{N}$  denote the set of non terminal nodes of the BDD, and let  $\mathcal{P}$  denote the set of all paths from the root to the terminal nodes of the BDD. Given a variable  $x$ , let  $\mathcal{N}(x)$  denote the set of nodes associated with  $x$ . Given a node  $u$ , let  $V(u)$  denote the variable associated with node  $u$ , let  $T(u)$  (or  $E(u)$ ) denote the node adjacent to  $u$  by the 1-edge (or 0-edge) outgoing from  $u$ , and let  $\mathcal{P}(u)$  denote all the paths from  $u$  to the terminal nodes. In a BDD, each variable has a *level*, and each node also has a level which is the same as its associated variable's level. In Fig. 1(a), the levels of variables  $a$ ,  $b$ , and  $c$  are 0, 1, and 2, respectively. Let's define them formally.

**DEFINITION 1 (VARIABLE LEVEL).** *The level of an input variable  $x$ ,  $l(x)$ , is defined as:  $l(x) = 0$  if  $x$  is a root variable;  $l(x) = \max\{l(V(u)) + 1 \mid u \in \mathcal{N} \wedge (x = V(T(u)) \vee x = V(E(u)))\}$  otherwise.*

**DEFINITION 2 (NODE LEVEL).** *The level of a non terminal node  $u$ ,  $l(u)$ , is defined as :  $l(u) = l(V(u))$ ; the level of a terminal node is the depth of the BDD.*

<sup>1</sup>For simplicity, we will not draw the arrows at the end of edges and the numbers by the edges in the rest of our paper.



**Figure 2:** Linear expansion of a BDD.

**DEFINITION 3 (CUT).** *Given a BDD, a cut at level  $i$  is a partition of the nodes so that all nodes with level less or equal to  $i$  belong to one side of the partition (upper side of cut  $i$ ), while the other nodes belong to the other side (lower partition of cut  $i$ ).*

**DEFINITION 4 (CUT SET).** *Given a BDD, the cut set at level  $i$  is the set of nodes from the lower side of cut  $i$  that have incoming edges from the upper side of cut  $i$ .*

**DEFINITION 5 (SUB-BDD).** *Given a BDD, the sub-BDD at node  $u \in \mathcal{N}$ , sub-BDD( $u$ ), is the BDD consisting of all nodes and edges reachable from  $u$  in the original BDD.*

For Fig. 1(a), sub-BDD( $v$ ) is shown in Fig. 1(b).

## 2.2 Linear Expansion

Node decomposition re-expresses a node function by a logically equivalent composition of two or more functions. Linear expansion is one such decomposition method. Fig. 2 shows the idea of linear expansion [16]. In Fig. 2(a),  $S = \{v_1, v_2, \dots, v_k\}$  is a cut set of the BDD; in Fig. 2(b), each BDD  $F_i$  is formed from  $F$  by replacing the nodes in  $S$  with terminal node 0 except the node  $v_i$ . It is easy to know that any path from the root to terminal node 1 must include one node from  $S$ . If the path includes the node  $v_i$ , then this path also appears in the BDD  $F_i$  in Fig. 2(b). This means all the paths from the root to terminal node 1 are covered by BDDs in Fig. 2(b). It is also easy to know that, for any BDD in Fig. 2(b), any path from the root to the terminal node 1 must be covered by the BDD in Fig. 2(a). So the BDD in Fig. 2(a) is equal to the summation of BDDs in Fig. 2(b). For each BDD in Fig. 2(b), we apply the *AND* decomposition to get Fig. 2(c). Fig. 3 shows an example of using linear expansion. The BDD in Fig. 3(a) is decomposed using linear expansion at cut2 (with cut set  $\{e, d, 1\}$ ), and Fig. 3(b) shows the decomposition. In this example,  $f_3 = 1$  is not shown in Fig. 3(b). For BDD  $c_3$ , node  $c$  is eliminated because both  $T(c)$  and  $E(c)$  are terminal node 0.

Even though the linear expansion is very powerful, we did not find previous synthesis algorithms actually using it. In this paper, we make the first attempt to synthesize circuits with linear expansion. We will introduce more definitions. In Fig. 2, the BDD  $F$  is decomposed into a set of small BDDs. And each small BDD, such as  $c_1$  or  $f_1$ , will also be decomposed using linear expansion for synthesis. The BDDs  $c_1, c_2, \dots, c_k$  are not sub-BDDs as defined in definition 5. Each of them is related to a root node, a cut level, and a cut set node.

**DEFINITION 6 (EXTENSION OF DEFINITION 4).** *Given a BDD, the cut set of  $u \in \mathcal{N}$  with regards to level  $i$ ,  $CS(u, i)$ , is the cut set of sub-BDD( $u$ ) at level  $i$ .*

In Fig. 3(a),  $CS(a, 0) = \{b, c\}$ ,  $CS(a, 2) = \{d, e, 1\}$ , and  $CS(a, 4) = \{1, 0\}$ .

**DEFINITION 7 (EXTENSION OF DEFINITION 5).** *Given a node  $u \in \mathcal{N}$ , a non negative integer  $i$ , and another node*

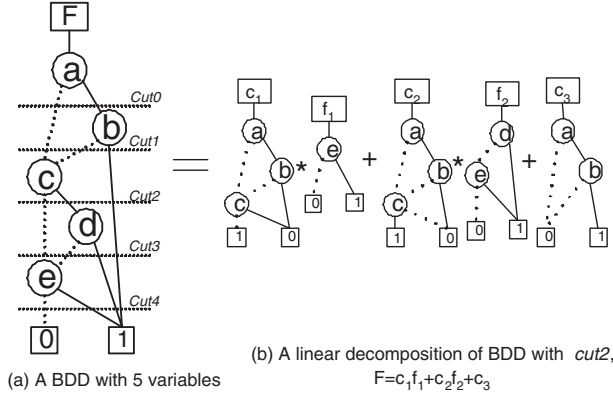


Figure 3: A linear decomposition example.

$v \in CS(u, i)$ , the sub-BDD rooted at  $u$  with respect to  $v$  at depth  $i$ ,  $\mathcal{B}_s(u, i, v)$ , is a modification of sub-BDD( $u$ ), where all nodes in the lower side of cut  $i$  except  $CS(u, i)$  are deleted from sub-BDD( $u$ ). All nodes in  $CS(u, i)$  are replaced to the terminal node 0 except the node  $v$ , which is replaced to the terminal node 1.

For the BDD in Fig. 3,  $c_1 = \mathcal{B}_s(a, 2, e)$ ,  $f_1 = \mathcal{B}_s(e, 0, 1)$ ,  $c_2 = \mathcal{B}_s(a, 2, d)$ ,  $f_2 = \mathcal{B}_s(d, 1, 1)$ , and  $c_3 = \mathcal{B}_s(a, 2, 1)$ . The BDD  $F$  in Fig. 3(a) is actually equal to the sub-BDD  $\mathcal{B}_s(a, 4, 1)$ . Particularly, a BDD is equal to its sub-BDD  $\mathcal{B}_s(r, n-1, 1)$ , where  $r$  is the root of the BDD, and  $n$  is the depth of the BDD.

---

#### Algorithm 1: Overall algorithm

---

**Input:** A Boolean network  
 $K$  : the input size of a LUT  
**Output:** Synthesized circuit

---

Collapse the Boolean network into a set of supernodes;  
Sort the nodes in a topological order from primary inputs to primary outputs;  
**foreach** node in order **do**  
  Collect the delay information of its fanins;  
  Perform our algorithm to synthesize the node;  
  Record the node delay;  
**end**

---

### 3. BDD SYNTHESIS FOR DELAY OPTIMIZATION

In this section, we present our BDD-based logic synthesis algorithm for delay optimization. Several key techniques are used, such as node clustering, linear expansion, dynamic programming, and various special decompositions. Algorithm 1 is a global overview of our algorithm. In the beginning, the nodes of the Boolean circuit are clustered and collapsed into supernodes based on node delays and whether there are gains to collapse them. The node collapsing reduces the number of nodes in the circuit, increases the functional complexity of each node, and gives us more room to optimize a node. After the node collapsing, we process supernodes in a topological order from primary inputs to primary outputs. Whenever we process a node, the delays of its fanins are known. Our dynamic programming algorithm uses delay information and linear expansion to synthesize the node to optimize its fanout delay. While we are processing a node, various special decompositions, such as XNOR and MUX decompositions, are also identified. The details will be presented in the following subsections.

#### 3.1 Clustering and Partial Collapsing

Clustering and partial collapsing is a critical step for a logic synthesis system. It can help removing logic redundancy, such as those caused by local reconvergence [9].

---

#### Algorithm 2: Clustering and partial collapsing algorithm

---

**Input:** A Boolean network  
**Output:** Partially collapsed circuit

---

**Local:**  $pq$  : a priority queue based on gains of merging two nodes, and it is in descending order of gains

---

```

begin
  done ← 0;
  repeat
    foreach fanin-fanout pair (in,out) do
      if mergable(in,out) then
        g = gain(in,out);
        pq.push(g,in,out);
      end
    if pq is empty then
      done ← 1;
    while pq is not empty do
      (g,in,out) ← pq.top();
      pq.pop();
      if either in or out is marked then
        jump to the next iteration;
      mark the node out;
      mergeBDD(in,out);
      if in has no fanouts then
        remove in from the network;
      end
    until done = 1
  end
end

```

---

In [17], the author used literal count numbers to guide the clustering; while in [9], BDS uses the number of BDD nodes as guidance. BDD-based collapsing method provides similar results compared to literal count method, but runs much faster [16]. The cost function in our algorithm considers both the number of BDD nodes and the node delays. Similar to previous approaches [4, 9], our clustering and partial collapsing algorithm is based on an iterative elimination framework.

Algorithm 2 shows how our partial collapsing method works. Basically, our algorithm runs for multiple iterations. In each iteration, a set of mergable node pairs are first collected, then nodes are merged in decreasing order of merging gains. The algorithm terminates when there is no feasible node pair that can be found. If a node is ever changed by a merging operation, the following merging operations regarding this node are cancelled in the current iteration. In the algorithm, the function  $mergable(in,out)$  tells us whether we should merge node  $in$  into node  $out$ . The function  $mergable$  first makes copies of BDDs of nodes  $in$  and  $out$ , then it merges copied BDDs. Let  $n$  denote the size of the merged BDD, and  $n_1, n_2$  denote the sizes of BDDs of nodes  $in$  and  $out$  before merging, respectively. The function  $mergable$  returns  $true$  only if  $n$  is smaller than a size bound (200 in our experiments) and  $n < (n_1 + n_2) * (1 + \alpha)$ , where  $\alpha$  is a parameter that can be adjusted. In this way, we do not merge two nodes if the BDD size after merging increases by a large portion. In the algorithm,

$$gain(x, y) = \begin{cases} (n_1 + n_2 - n) * (1 + \beta * \frac{d_o(x)}{d_{ix}(y)} + \gamma/n_o(x)), & \text{if } n_1 + n_2 \geq n \\ (n_1 + n_2 - n)/(1 + \beta * \frac{d_o(x)}{d_{ix}(y)} + \gamma/n_o(x)), & \text{if } n_1 + n_2 < n \end{cases}$$

where  $d_o(x)$  is the output delay of node  $x$ ,  $d_{ix}(y)$  is the maximum delay of fanins of node  $y$ ,  $n_o(x)$  is the number of fanouts of node  $x$ , and  $\beta, \gamma$  are user controlled parameters. For simplicity, we use  $x$  and  $y$  to represent  $in$  and  $out$ , respectively. From the formula, the larger the delay of a fanin node, the larger the gain of merging it with its fanouts. Another heuristic rule used in our algorithm is to give a higher preference to a fanin node with smaller number of fanouts because a node can be removed from the network if it has no fanouts after merging. The smaller the fanout

---

**Algorithm 3:** Logic synthesis algorithm for one BDD

---

**Input:**  $inputDelay$  :  $inputDelay(x)$  is the delay of the input variable  $x$

**Output:** Synthesized network and its delay

---

**Local:**  $tmpDelay, bestDelay$  : temporary variables

---

**Global:**  $delay$  :  $delay(\mathcal{B}_s(u, l, v))$  is the delay of sub-BDD  $\mathcal{B}_s(u, l, v)$ ;

---

**begin**

reduce the size of the BDD by a reordering algorithm;

$n \leftarrow$  number of input variables of the BDD;

$\mathcal{N} \leftarrow$  set of all BDD nodes;

**for**  $l = 0$  **to**  $n - 1$  **do**

**foreach**  $u \in \mathcal{N}$  **do**

**if**  $l(u) + l > n - 1$  **then**

      | jump to the next iteration;

**enumerateCS**( $u, l$ );

**foreach**  $v \in CS(u, l)$  **do**

$bestDelay \leftarrow +\infty$ ;

**if**  $l = 0$  **then**

$bestDelay \leftarrow inputDelay(V(u))$ ;

**for**  $j = 0$  **to**  $l - 1$  **do**

$tmpDelay \leftarrow$

$delayDecompose(u, l, v, j)$ ;

**if**  $tmpDelay < bestDelay$  **then**

$bestDelay \leftarrow tmpDelay$ ;

**end**

$delay(\mathcal{B}_s(u, l, v)) \leftarrow bestDelay$ ;

**end**

**end**

**end**

produce the network using the linear decomposition

based on the cuts we choose for all sub-BDDs;

set the network delay to be  $delay(\mathcal{B}_s(r, n - 1, 1))$ ,

where  $r$  is the root of the BDD;

**end**

---

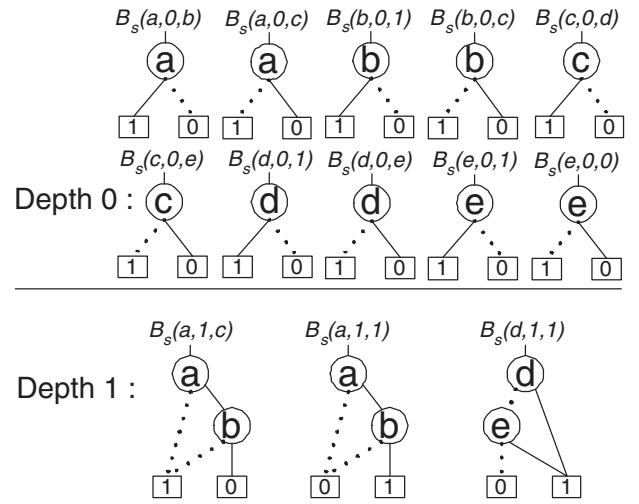
number, the less duplication it incurs due to merging. The function  $mergeBDD(in, out)$  in the algorithm merges node  $in$  into node  $out$ , and removes the fanin and fanout relation between them.

### 3.2 Dynamic Programming Algorithm to Synthesize one BDD for Delay Optimization

In this section, we present how to synthesize a super-node of the collapsed network. Our algorithm uses a BDD to represent this node, and decomposes the BDD recursively to form a Boolean network. During the decomposition, our algorithm tries to optimize the network delay. The algorithm produces both the decomposed (synthesized) network and its delay. Let's look at an example first. For the BDD in Fig. 3(a), there are 5 possible cuts, and each cut produces a different synthesis result. In our algorithm, we try all these 5 cuts, and choose the one producing the smallest delay. Fig. 3(b) shows the decomposition produced by the cut2 in Fig. 3(a). Obviously, we need to know the delays of sub-BDDs before we can calculate the delay of the BDD  $F$  in terms of cut2. To get the delays of sub-BDDs, we recursively apply our dynamic programming algorithm. Actually, the algorithm starts with the smallest sub-BDDs, and processes sub-BDDs in an increasing order of their depths. (In this paper, the depth and level for a BDD are interchangeable with each other.)

#### 3.2.1 Main Algorithm

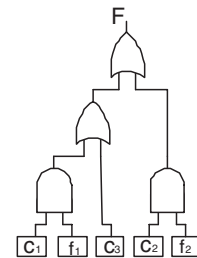
Algorithm 3 shows our dynamic programming-based algorithm. The size of the BDD is first minimized by using a BDD reordering algorithm [15]. After reordering, the algorithm processes sub-BDDs in an increasing order of their depths, from depth 0 to depth  $n - 1$ . For each depth  $l$ , the algorithm visits all BDD nodes; for each such BDD node  $u$ , the depth of sub-BDD( $u$ ) is  $n - l(u)$ , and its maximum possible cut level is  $n - l(u) - 1$ , so the algorithm generates the cut set



**Figure 4:** This figure enumerates all depth-0 sub-BDDs of the BDD in Fig. 3(a), and 3 depth-1 sub-BDDs.

$CS(u, l)$  by the procedure  $enumerateCS$  (see algorithm 4) only if  $l \leq n - l(u) - 1$ . For each cut node  $v \in CS(u, l)$ , the algorithm produces a sub-BDD  $\mathcal{B}_s(u, l, v)$ . For the sub-BDD  $\mathcal{B}_s(u, l, v)$ , the algorithm tries all cuts from 0 to  $l - 1$ , produces a delay for each cut (by calling the function  $delayDecompose$ ), and chooses the smallest delay as the delay of  $\mathcal{B}_s(u, l, v)$ . Fig. 4 enumerates all depth-0 sub-BDDs of the BDD in Fig. 3(a), and 3 depth-1 sub-BDDs. Whenever the algorithm processes a depth  $i$  sub-BDD, all sub-BDDs with depths less than  $i$  have been processed, so all the information needed for decomposing the depth  $i$  sub-BDD is known. For example, in Fig. 3(b), the depth of sub-BDD  $f_1$  is 0; the depth of  $f_2$  is 1; and the depths of  $c_1$ ,  $c_2$  and  $c_3$  are all 2. So when the algorithm starts to process BDD  $F$  in Fig. 3(a), the sub-BDDs it is decomposed into at cut2 have already been processed, so do sub-BDDs for other cuts of  $F$ . We can use these sub-BDDs to decide which cut is the best for decomposing  $F$  and the corresponding delay. After getting optimum cuts for all sub-BDDs, it is easy to produce the synthesized network using linear expansion. We do not show the details of this part (see Fig. 5 for an example of constructing Boolean network from a BDD decomposition). Since  $\mathcal{B}_s(r, n - 1, 1)$  is actually the BDD of the supernode,  $delay(\mathcal{B}_s(r, n - 1, 1))$  is the delay of the synthesized network.

Algorithm 4 produces the cut set  $CS(u, l)$ . If  $l = 0$ , the cut set is the set of nodes adjacent to  $u$  joined by the edges outgoing from  $u$ , so  $CS(u, l) = \{T(u), E(u)\}$ . If  $l > 0$ , the cut set  $CS(u, l)$  can be constructed from the cut set  $CS(u, l - 1)$  as shown in the algorithm. For a cut node  $v \in CS(u, l - 1)$ , if  $l(v) > l(u) + l$ , the node  $v$  is at the lower side of cut  $l$  for sub-BDD( $u$ ), so  $v \in CS(u, l)$ ; otherwise,  $T(v) \in CS(u, l)$  and  $E(v) \in CS(u, l)$ . In Fig. 3(a), for



**Figure 5:** This figure shows the Boolean network generated according to linear expansion at cut2 for the BDD in Fig. 3. The nodes  $c_1$ ,  $f_1$ ,  $c_2$ ,  $f_2$ ,  $c_3$  represent corresponding Boolean networks of sub-BDDs.

example,  $CS(a, 0) = \{b, c\}$ .  $l(c) = 2 > l(a) + 1 = 1$  implies  $c \in CS(a, 1)$ , and  $l(b) = l(a) + 1$  implies  $T(b) = 1 \in CS(a, 1)$  and  $E(b) = c \in CS(a, 1)$ . So,  $CS(a, 1) = \{c, 1\}$ .

---

**Algorithm 4:** enumerateCS

---

**Input:**  $u$  : a BDD node  
 $l$  : the cut level  
**Output:**  $CS(u, l)$

---

```

begin
  if  $l=0$  then
     $CS(u, l) \leftarrow \{T(u), E(u)\}$ ;
  else
     $CS(u, l) \leftarrow \emptyset$ ;
    foreach  $v \in CS(u, l-1)$  do
      if  $l(u) + l < l(v)$  then
         $CS(u, l) \leftarrow CS(u, l) \cup \{v\}$ ;
      else
         $CS(u, l) \leftarrow CS(u, l) \cup \{T(v), E(v)\}$ ;
      end
    end
  end
end
end

```

---

### 3.2.2 Producing the Delay for a sub-BDD

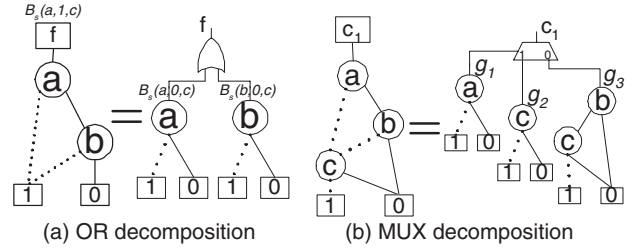
The function *delayDecompose* in algorithm 3 first generates a set of AND gates with known delays from the decomposition, which are all fanins of an OR gate, then it uses bin packing algorithm in [18] to decompose the OR gate and produces the delay of the decomposition (see Fig. 5). Given a sub-BDD  $\mathcal{B}_s(u, l, v)$ , and a cut at  $j$ , the linear expansion decomposes the sub-BDD into a summation of several AND gates. Each AND gate corresponds to a node  $w \in CS(u, j)$ , and the inputs of this AND gate are sub-BDDs  $\mathcal{B}_s(u, j, w)$  and  $\mathcal{B}_s(w, l(u) + l - l(w), v)$  (please refer to Fig. 3(b)), and the input delay of the AND gate is the maximum delay of its two inputs. If  $w = v$ , then the AND gate is degenerated to have only one input:  $\mathcal{B}_s(u, j, v)$ , and the other input is eliminated because it is logic const 1.

### 3.2.3 Special Decompositions

In our algorithm, we also check conditions under which various special decompositions can be applied. If any one of these conditions is satisfied, the corresponding special decomposition is applied instead of the linear expansion. We prefer special decompositions to linear expansion for these cases because these special decompositions use less sub-BDDs during decomposition. For example, OR decomposition uses 2 sub-BDDs instead of 3 used by linear expansion. In this section,  $\mathcal{B}_s(u, l, v)$  and  $j$  have the same meaning as in the previous sections. The conditions under which  $\mathcal{B}_s(u, l, v)$  has special decompositions are listed as the following:

- *AND decomposition*: this is a special case of linear expansion, where there is only one AND gate.
- *OR decomposition*<sup>2</sup>: the condition for an OR decomposition is that  $|CS(u, j)| = 2$  and  $v \in CS(u, j)$ . Let us assume  $CS(u, j) = \{v, w\}$ . In this case,  $\mathcal{B}_s(u, l, v)$  can be decomposed as  $\mathcal{B}_s(u, j, v) \vee \mathcal{B}_s(w, l(u) + l - l(w), v)$ . In Fig. 6(a),  $\mathcal{B}_s(a, 1, c)$  is a sub-BDD of the BDD in Fig. 3(a). Since  $CS(a, 0) = \{b, c\}$ ,  $\mathcal{B}_s(a, 1, c)$  has an OR decomposition at cut 0, and it is decomposed as  $\mathcal{B}_s(a, 0, c) \vee \mathcal{B}_s(b, 0, c)$ .
- *MUX decomposition*: the condition for an MUX decomposition is  $|CS(u, j)| = 2$ . Let  $CS(u, j) = \{w_1, w_2\}$ .

<sup>2</sup>Even though the OR decomposition can be identified by finding the AND decomposition of the complemented BDD (DeMorgan's rule), it is much easier to find the OR decomposition according to its structural property instead of complementing every sub-BDD and figuring out the AND decomposition.



**Figure 6: Special decompositions.** For the BDD in Fig. 3, (a) the OR decomposition can be applied to sub-BDD  $\mathcal{B}_s(a, 1, c)$  at cut 0, and  $\mathcal{B}_s(a, 1, c) = \mathcal{B}_s(a, 0, c) \vee \mathcal{B}_s(b, 0, c)$ ; (b) the MUX decomposition can be applied to sub-BDD  $c_1 = \mathcal{B}_s(a, 2, e)$  at cut 0, and  $c_1 = (g_1 \wedge g_2) \vee (\bar{g}_1 \wedge g_3)$ , where  $g_1 = \mathcal{B}_s(a, 0, c)$ ,  $g_2 = \mathcal{B}_s(c, 0, e)$ , and  $g_3 = \mathcal{B}_s(b, 1, e)$ .

In this case,  $\mathcal{B}_s(u, l, v)$  can be decomposed as  $(\mathcal{B}_s(u, j, w_1) \wedge \mathcal{B}_s(w_1, l(u) + l - l(w_1), v)) \vee (\neg \mathcal{B}_s(u, j, w_1) \wedge \mathcal{B}_s(w_2, l(u) + l - l(w_2), v))$ . In Fig. 3,  $CS(a, 0) = \{b, c\}$ , so the sub-BDD  $\mathcal{B}_s(a, 2, e)$  has an MUX decomposition at cut 0, and it is decomposed as  $(\mathcal{B}_s(a, 0, c) \wedge \mathcal{B}_s(c, 0, e)) \vee (\neg \mathcal{B}_s(a, 0, c) \wedge \mathcal{B}_s(b, 1, e))$  [Fig. 6(b)].

- *XNOR decomposition*: the condition for an XNOR decomposition is that  $|CS(u, j)| = 2$  and the Boolean functions of  $w_1, w_2$  are complement to each other, where  $CS(u, j) = \{w_1, w_2\}$ . In this case,  $\mathcal{B}_s(u, l, v)$  can be decomposed as  $\mathcal{B}_s(u, j, w_1) \oplus \mathcal{B}_s(w_1, l(u) + l - l(w_1), v)$ . XNOR decomposition is a special case of MUX decomposition.

## 3.3 Complexity of the Algorithm

Let  $n$  denote the number of input variables, and  $N$  denote the size of the BDD. The runtime of algorithm 3 is summarized by the following theorem. Since the BDD size is limited in our algorithm (up to 200 in our experiments), and the number of input variables is small (less than 20 for most cases), the algorithm is fast.

**THEOREM 1.** *If the size of a BDD is  $N$ , and it has  $n$  input variables, then the runtime of synthesizing the BDD is  $O(n^2 N^2)$ , and the algorithm uses  $O(nN^2)$  memory space.*

## 4. EXPERIMENTAL RESULTS

We compare our algorithm, named DDBDD, with BDS-pga. In the experiments, we use both 10 largest combinatorial MCNC benchmark circuits (BDS-pga has limited support for the 10 largest MCNC sequential benchmark circuits) and the benchmark circuits provided by the authors of BDS-pga [10]. The input size  $K$  of LUTs is 5, and the depth is the number of LUT levels in the final mapped circuit. We perform our experiments on a desktop PC with a 3GHz Intel CPU, and the operating system is Red Hat Linux 8.0.

Table 1 shows the comparison of DDBDD and BDS-pga on ten largest MCNC combinatorial benchmarks. The circuits are first mapped by both DDBDD and BDS-pga. We then feed the circuits into VPR [19] to run placement and routing. We use a cluster size 10 and length 4 wire segments in the experiment. We first run VPR to obtain routing results with the minimum number of routing tracks for each circuit, and then apply additional 20% routing tracks and rerun routing to get the final results, as commonly practiced [19]. The maximum delay of each circuit is collected. Table 1 shows BDS-pga produces circuits with 95% more mapped network depth (or 20% more delay after placement and routing) with 22% less area. The large runtime for DDBDD is due to the collapsing procedure, which runs for many iterations, and we will reduce the runtime in our future work. Table 2 shows the comparison on the set of smaller circuits used in [10]. In the table, the data of BDS-pga are from [20]. In this set of circuits, BDS-pga produces circuits

Table 1: The column Depth shows the maximum level of the mapped circuit, the column LUTs shows the number of LUTs, the column Delay shows the maximum delay according to VPR, and the column Runtime is the runtime of the program. In columns Comparison, each field shows the value of the BDS-pga result divided by the DDBDD result.

Ckt.	DDBDD				BDS-pga				Comparison		
	Depth	LUTs	Delay(s)	Runtime(s)	Depth	LUTs	Delay(s)	Runtime(s)	Depth	LUTs	Delay
alu4	7	1244	1.05E-08	47.72	11	1116	1.37E-08	5.2	1.57	0.90	1.30
apex2	8	1604	1.28E-08	101.38	14	1317	1.49E-08	4.5	1.75	0.82	1.16
apex4	6	1314	1.03E-08	100.92	15	991	1.32E-08	5.5	2.5	0.75	1.28
des	6	1552	1.08E-08	49.86	7	1106	1.18E-08	4.6	1.17	0.71	1.09
ex1010	7	4456	1.55E-08	548.37	20	3636	1.92E-08	1078	2.86	0.82	1.24
ex5p	6	1381	1.19E-08	169.63	11	810	1.31E-08	2.6	1.83	0.59	1.10
misex3	6	1224	1.03E-08	50.48	10	1032	1.10E-08	4.2	1.67	0.84	1.07
pdcc	8	4042	1.47E-08	379.77	18	3210	2.17E-08	31	2.25	0.79	1.48
seq	6	1518	9.59E-09	79.47	12	1214	1.13E-08	6.3	2	0.80	1.18
spla	8	3243	1.54E-08	1483.95	15	2659	1.76E-08	25.3	1.88	0.82	1.14
average				301.16				116.72	1.95	0.78	1.20

Table 2: Comparison Results of DDBDD with BDS-pga.

Ckt.	DDBDD		BDS-pga		Comparison	
	depth	LUTs	depth	LUTs	depth	LUTs
5xp1	2	19	2	16	1	0.84
9sym	3	8	3	8	1	1
9symml	3	8	3	8	1	1
alu2	4	70	5	45	1.25	0.64
apex6	4	249	6	194	1.5	0.78
apex7	3	113	5	69	1.67	0.61
b9	3	55	3	43	1	0.78
c1355	4	72	4	66	1	0.92
c1908	7	228	9	118	1.29	0.52
c499	4	71	4	65	1	0.92
c5315	6	553	8	445	1.33	0.80
c880	7	193	9	123	1.29	0.64
clip	3	32	5	42	1.67	1.31
count	3	50	5	34	1.67	0.68
duke2	3	208	7	180	2.33	0.87
misex1	2	14	2	14	1	1
rd84	3	16	3	14	1	0.875
rot	6	301	10	223	1.67	0.74
t481	2	5	2	5	1	1
vg2	4	80	5	61	1.25	0.76
Average					1.30	0.83

with 30% more mapped network depth and 17% less area on average compared to our algorithm. Therefore, we show that our solution provides a significant amount of performance gain, which makes the trade off on area worthwhile if the design goal is for high performance.

## 5. CONCLUSIONS AND ACKNOWLEDGEMENTS

In this work, we presented a BDD-based logic synthesis algorithm to optimize the performance of FPGA designs. We carried out gain-based circuit collapsing and dynamic programming-driven BDD decomposition to minimize circuit delay. BDD decomposition was mainly carried out through linear expansion and further enhanced by special decomposition cases when necessary. Our algorithm was delay-centric overall. Especially, the dynamic programming approach was designed to efficiently search through all the possible decompositions in a BDD to achieve the minimal delay among these decompositions. We showed that we were able to achieve a significant amount of performance gain with relatively smaller area overhead compared to a previous state-of-art BDD synthesis algorithm for FPGAs. The future work is to further reduce area and runtime. One direction is performance/area/runtime tradeoff.

We would like to thank professor Russell Tessier's group for providing the benchmarks. This work is partially spon-

sored by Altera Corporation through a research grant. We used machines donated by Intel.

## 6. REFERENCES

- [1] R. K. Brayton, G. D. Hachtel, et al. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
- [2] H. Savoj, R. K. Brayton, and H. Touati. Extracting Local Dont Cares for Network Optimization. In *ICCAD*, pages 514–517, 1991.
- [3] H. Savoj and R. K. Brayton. The Use of Observability and External Dont Cares for the Simplification of Multi-Level Networks. In *DAC*, pages 297–301, 1991.
- [4] E. Sentovich, K. Singh, et al. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL Memorandum M89/49, Department of EECS, University of California, Berkeley, May 1992.
- [5] K. C. Chen, J. Cong, Y. Ding, A. B. Kahng, and P. Trajmar. DAG-Map: Graph-Based FPGA Technology Mapping for Delay Optimization. In *IEEE Des. Test Comput.*, pages 7–20, 1992.
- [6] D. Chen and J. Cong. DAOmap: A Depth-optimal Area Optimization Mapping Algorithm. In *ICCAD*, pages 752–759, 2004.
- [7] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Trans. on CAD*, 13(1):1–12, 1994.
- [8] A. Mishchenko, S. Chatterjee, and R. Brayton. Improvements to technology mapping for LUT-based FPGAs. In *FPGA*, pages 41–49, 2006.
- [9] C. Yang and M. Ciesielski. BDS: A BDD-Based Logic Optimization System. *IEEE Trans. on CAD*, 21(7):866–876, 2002.
- [10] N. Vemuri, P. Kalla, and R. Tessier. BDD-based Logic Synthesis for LUT-based FPGAs. *IEEE Trans. on DAES*, 7:501–525, 2000.
- [11] V. Manoharajah, D. P. Singh, and S. D. Brown. Post-Placement BDD-Based Decomposition for FPGAs. In *FPL*, pages 31–38, 2005.
- [12] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, 35:677–691, 1986.
- [13] C. Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38(4):985–999, 1959.
- [14] S. B. Akers. Functional Testing with Binary Decision Diagrams. In *Eighth Annual Conf. on Fault Tolerant Computing*, pages 75–82, 1978.
- [15] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, pages 42–47, 1993.
- [16] C. Yang. BDD-Based Logic Synthesis System. Ph.D thesis, EECS Department, Univ. of Massachusetts, Amherst, 2000.
- [17] R. Rudell. Logic Synthesis for VLSI Design. Ph.D thesis, EECS Department, Univ. of California, Berkeley, 1989.
- [18] R. J. Francis, J. Rose, and Z. G. Vranesic. Technology Mapping Lookup Table-based FPGAs for Performance. In *ICCAD*, pages 568–571, 1991.
- [19] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.
- [20] [Online]. Available: [http://www.ecs.umass.edu/ece/tessier/rcg/bds-pga-2.0/results\\_bds-pga.html](http://www.ecs.umass.edu/ece/tessier/rcg/bds-pga-2.0/results_bds-pga.html).