DDBDD: Delay-Driven BDD Synthesis for FPGAs

Lei Cheng, Deming Chen, and Martin D. F. Wong, *Fellow, IEEE*

***Abstract*—In this paper, we target field-programmable gate array (FPGA) performance optimization using a novel binary decision diagram (BDD)-based synthesis paradigm. Most previous works have focused on BDD size reduction during logic synthesis. In this paper, 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 called 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; and 3) we consider special decomposition scenarios coupled with linear expansion for further improvement on the quality of results. Experimental results show that we can achieve a 30% performance gain with a 22% area overhead on the average compared to a previous state-of-the-art BDD-based FPGA synthesis tool, namely, BDS-pga. Compared to DAOmap, we can achieve a 33% performance gain with only an 8% area overhead. Compared to the ABC mapper, we can achieve a 20% performance gain with only an 8% area overhead.**

***Index Terms*—Binary decision diagram (BDD), field-programmable gate array (FPGA), logic decomposition.**

I. Introduction

THE field-programmable gate array (FPGA) has become increasingly popular throughout the past decade. The cost pressures, changing requirements, and short design windows favor increasingly more programmable chip solutions. An FPGA chip consists of programmable logic blocks, programmable interconnections, and programmable input/output pads. The lookup table (LUT)-based FPGA architecture dominates the existing programmable chip industry, in which the basic programmable logic element is a K-input LUT (K-LUT). A K-LUT can implement any Boolean function of up to K variables. Similar to the application-specified integrated circuit design flow, the FPGA design process consists of the system-level design, the logic synthesis, and the physical design. Many algorithms have been proposed to facilitate the automatic design process of FPGAs to improve performance, area, or power. In this paper, we present a new logic synthesis algorithm for FPGAs for circuit delay reduction, which is based on binary decision diagrams (BDDs).

Manuscript received May 29, 2007; revised September 7, 2007 and November 26, 2007. This work was supported in part by Altera Corporation under a research grant. This paper was recommended by Associate Editor S. Nowick.

L. Cheng is with Synplicity, Inc., Sunnyvale, CA 94086 USA (e-mail: lcheng1@uiuc.edu).

D. Chen and M. D. F. Wong are with the Department of Electrical and Computer Engineering and Coordinated Science Laboratory, University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA.

Digital Object Identifier 10.1109/TCAD.2008.923088

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], which is 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]–[10] 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.

Although the traditional logic optimization methodology is very successful on AND/OR-intensive circuits, its performance on XOR-intensive circuits is far from satisfactory (see [11]). In [11], the authors presented BDS, a logic optimization system based on BDD decomposition techniques. By exploring the structure of a BDD, BDS is able to identify not only AND/OR decompositions, but also XOR/multiplexer (MUX) decompositions. BDS has been successfully applied to FPGA designs in [12]. The logic synthesis system presented in [12], namely, BDS-pga, first collapses the network using a maximum fanout-free cone (MFFC)-based eliminating method and then recursively cuts BDDs in the middle 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 [12] compared to SIS + Flowmap [8]. Another BDD-based synthesis technique is introduced in [13], where BDD resynthesis is applied to improve timing. After placement, some timing critical parts of the circuit are selected, resynthesized, and then replaced.

One of the drawbacks of BDS, as mentioned in [11], is its inability to consider delay optimization, because it cannot properly balance the factoring tree used in their algorithm. To optimize the delay, BDS-pga uses a delay resynthesis approach. After logic synthesis, BDS-pga finds out critical paths and partially collapses these paths. Then, it uses the ESPRESSO [1] algorithm to optimize the collapsed nodes and redecomposes 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. The decomposition method used in [13] is based on a timing-driven reordering heuristic.

Although it is useful, this heuristic does not show a significant improvement on the experimental results (6%, as mentioned in their paper).

In this paper, we propose a BDD-based FPGA logic synthesis system, called the delay-driven BDD (DDBDD), targeting delay optimization. We use the unit delay model in our algorithm. The depth of the mapped circuit is used to estimate the delay or performance of the mapped circuit. The *depth* of a node in a circuit is the maximum number of nodes between this node and one primary input. The depth of a primary input is zero. The *mapping depth* of a node is the depth of the LUT covering this node in the mapped circuit. We first introduce a gain-based partial collapsing algorithm that considers delay, and then, we present a dynamic programming algorithm for synthesizing each collapsed node to optimize the delay. Our algorithm uses BDDs to represent node functions, and it uses linear expansion for BDD decomposition, which is a generalized decomposition technique of all the different decompositions used in BDS, to synthesize the circuit. Based on linear expansion, our dynamic programming algorithm chooses 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 the quality of results. Experimental results show that we can achieve a performance gain of up to 30% with a 22% area overhead compared to BDS-pga. Compared to DAOmap [6], we can achieve a 33% performance gain with only an 8% area overhead. Compared to the ABC mapper [7], we can achieve a 20% performance gain with only an 8% area overhead.

In Section II, we introduce some related terminologies and preliminaries. Section III presents our algorithm in detail. Experimental results are shown in Section IV, and we conclude this paper in Section V.

II. Definitions and Preliminaries

It is assumed that readers are familiar with the basic concepts of Boolean functions, Boolean networks [14], and BDDs [15]. We provide a brief review of related concepts and define several terminologies used in this paper.

A. 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, which is 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. For complete definitions of these terms, we refer readers to [14]. In this paper, the term *Boolean function* is used for a completely specified Boolean function.

BDDs were first introduced by Lee [16] and then popularized by Akers [17]. In [15], 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. Formally, a BDD is a DAG, representing a Boolean function, with two terminal nodes 1 and 0. Each nonterminal node has an index to identify an input variable of the Boolean function and has two outgoing edges, which are called the 0-edge and the 1-edge. In the BDD drawings in this paper, we use solid lines to represent 1-edges and dotted lines to represent 0-edges. The depth 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 to ROBDDs as BDDs. Fig. 1(a) is a BDD example.[1] 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 [18].

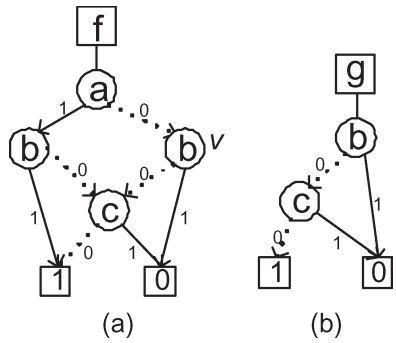


Fig. 1. (a) BDD for the Boolean function: $f = a \cdot b \vee \bar{b} \cdot \bar{c}$, and the variable inside each node is the input variable associated with the node. (b) Sub-BDD(v) of the BDD in (a).

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 nonterminal 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 us define them formally.

Definition 1 (Variable Level): The level of an input variable x, i.e., $l(x)$, is defined as $l(x) = 0$ if x is a root variable and $l(x) = \max\{l(V(u)) + 1 | u \in \mathcal{N} \wedge (x = V(T(u)) \vee x = V(E(u)))\}$ otherwise.

Definition 2 (Node Level): The level of a nonterminal node u, i.e., $l(u)$, is defined as $l(u) = l(V(u))$. The level of a terminal node is the depth of the BDD.

Definition 3 (Cut): Given a BDD, a cut at level i is a partition of the nodes so that all nodes with a level less than or equal to i belong to one side of the partition (upper side of cut i), whereas 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.

[1]For simplicity, we will not draw the arrows at the end of the edges and the numbers by the edges indicating their types (0-edge or 1-edge) in the rest of our paper.

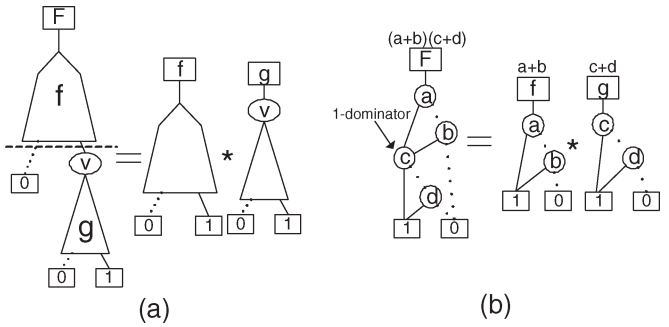


Fig. 2. Algebraic AND decomposition with 1-dominator. (a) Principle of algebraic AND decomposition with 1-dominator. Node v is a 1-dominator, and the Boolean function F is decomposed as $F = f \cdot g$. (b) Real circuit using algebraic AND decomposition.

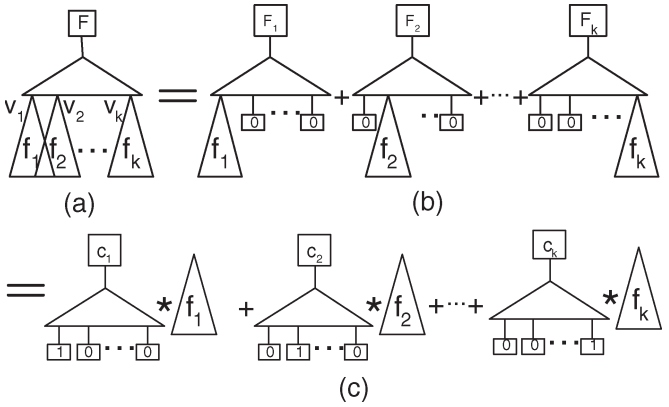


Fig. 3. Linear expansion of a BDD. (a) Generic BDD. (b) Linear expansion of the BDD. (c) Decomposition of all components using 1-denominator.

Definition 5 (Sub-BDD): Given a BDD, the sub-BDD at node $u \in \mathcal{N}$, i.e., sub-BDD(u), is the BDD consisting of all nodes and edges that are reachable from u in the original BDD.

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

B. Functional Decomposition

Node decomposition reexpresses a node function by a logically equivalent composition of two or more functions. In this section, we introduce several BDD-based decomposition techniques that are used in this paper. In [19], Karplus introduced the 1-dominator and the 0-dominator. Basically, in a BDD without complement edges, a 1-dominator (0-dominator) is a node that belongs to every path from the root to terminal node 1 (0). A 1-dominator (0-dominator) leads to an algebraic AND (OR) decomposition (see Fig. 2 for an example of a 1-dominator). In [11], Yang and Ciesielski introduced the concept of the x-dominator. In a BDD with complement edges, an x-dominator is a node that is contained in every path from the root to the terminal nodes. As shown in [11], an x-dominator corresponds to an XNOR decomposition. In [11], it was also shown that if a BDD has two nodes covering all paths from the root to the terminal nodes, then it has a MUX decomposition. All these decompositions are special cases of linear expansion [20]. Fig. 3 shows the idea of linear expansion. In Fig. 3(a), $S = \{v_1, v_2, \ldots, v_k\}$ is a cut set of the BDD; in Fig. 3(b), each BDD F_i is formed from F by replacing the nodes in S to terminal node 0, except the node v_i. Obviously, 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. 3(b). This means that all the paths from the root to terminal node 1 are covered by BDDs in Fig. 3(b). It is also easy to know that for any BDD in Fig. 3(b), any path from the root to terminal node 1 must be covered by the BDD in Fig. 3(a). Therefore, the BDD in Fig. 3(a) is equal to the summation of BDDs in Fig. 3(b). For each BDD in Fig. 3(b), we apply the AND decomposition to get Fig. 3(c). In our algorithm, we try linear expansions on all possible cuts for a BDD using a dynamic programming algorithm and pick the cut that produces the best result. Fig. 4 shows a BDD with five cuts, and Fig. 5 shows an example of applying linear expansion at cut2. The BDD in Fig. 5(a) is decomposed using linear expansion at cut2 (with cut set $\{e, d, 1\}$), and Fig. 5(b) shows the decomposition. In this example, $f_3 = 1$ is not shown in Fig. 5(b). For BDD c_3, node c is eliminated because both $T(c)$ and $E(c)$ are terminal node 0.

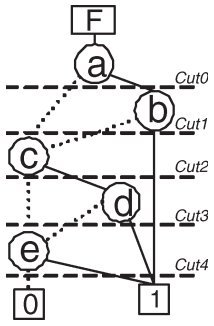


Fig. 4. BDD example with five cuts.

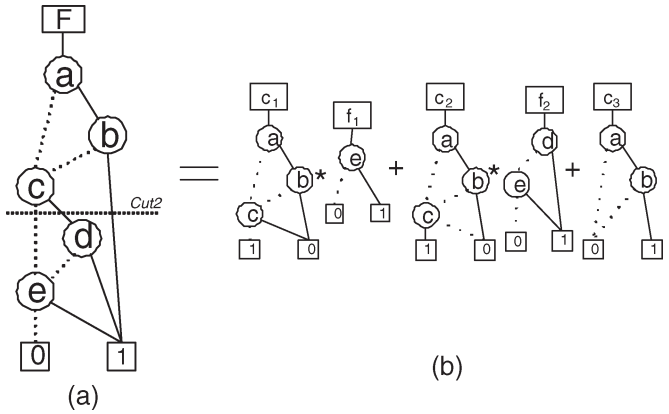


Fig. 5. Linear decomposition example. (a) BDD with five variables. (b) Linear decomposition of BDD with cut2.

Even though the linear expansion is very powerful, we did not find previous synthesis algorithms using it. In this paper, we make the first attempt to synthesize circuits with linear expansion. We will introduce more definitions. In Fig. 3, the BDD F is decomposed into a set of small BDDs. Furthermore, each small BDD, such as c_1 or f_1, will also be decomposed using linear expansion for synthesis. The BDDs $c_1, c_2, \ldots, c_k$ are not sub-BDDs, as defined in Definition 5. Each of them is actually 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 regard to level i, i.e., $CS(u, i)$, is the cut set of sub-BDD(u) at level i.

In Fig. 5(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 nonnegative integer i, and another node $v \in CS(u,i)$, the sub-BDD rooted at u with respect to v at depth i, i.e., $\mathcal{B}_s(u,i,v)$, 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 terminal node 0, except the node v, which is replaced to terminal node 1.

For the BDD in Fig. 5, $c_1=\mathcal{B}_s(a,2,e)$, $f_1=\mathcal{B}_s(e,0,1)$, $c_2=\mathcal{B}_s(a,2,d)$, $f_2=\mathcal{B}_s(d,1,1)$, and $c_3=\mathcal{B}_s(a,2,1)$. The BDD F in Fig. 5(a) is equal to the sub-BDD $\mathcal{B}_s(a,4,1)$. Generally, 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.

III. 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 dominator-based 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 depths 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 the supernodes in a topological order from the primary inputs to the primary outputs. Whenever we process a node, the mapping depths of its fanins are known. Our dynamic programming algorithm uses mapping depth information and linear expansion to synthesize the node to optimize its fanout mapping depth. 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 sections.

Algorithm 1: Overall algorithm
Input: A Boolean network
K: the input size of an LUT
Output: Synthesized circuit
Collapse the Boolean network into a set of supernodes using Algorithm 2;
Sort the nodes in a topological order from primary inputs to primary outputs;
for each *node in order* **do**
  Collect the mapping depth information of its fanins;
  Perform Algorithm 3 to synthesize the node;
  Record the node minimum mapping depth;
end

A. Clustering and Partial Collapsing

Clustering and partial collapsing is a critical step for a logic synthesis system. It can help remove logic redundancies, such as those caused by local reconvergence [11]. Similar to previous approaches [4], [11], our clustering and partial collapsing algorithm is based on an iterative elimination framework.

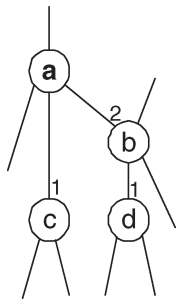


Fig. 6. In this example, both gates b and c fan out to gate a, so a is the *out* gate, and b and c are the *in* gates in Algorithm 2. The output depths of gates b and c are 2 and 1, respectively. We prefer to merge b into a instead of merging c into a, because it is more likely to reduce the output depth of a.

In [4], SIS uses the number of literal counts as the cost function to guide the clustering; whereas in [11], BDS uses the number of BDD nodes as guidance. The BDD-based collapsing method provides results that are similar to the literal counting method but runs much faster [20]. The cost function in our algorithm considers both the number of BDD nodes and the node depths.

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 is 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 changed by a merging operation, the following merging operations regarding this node are canceled 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*, and then, it merges copied BDDs. Let n denote the size of the merged BDD and n_1 and n_2 denote the sizes of the 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 α is a parameter that can be adjusted, and $\alpha=3$ in our experiments. This way, we do not merge two nodes if the BDD size after merging increases by a large portion. In the algorithm, we define

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

where $d_o(x)$ is the output depth of node x, $d_{ix}(y)$ is the maximum depth of fanins of node y, $n_o(x)$ is the number of fanouts of node x, and β and γ are user-controlled parameters. For simplicity, we use x and y to represent *in* and *out*, respectively. From the formula, the larger the depth $d_o(x)$ of a fanin node x is, the larger the gain of merging it with one of its fanouts will be. The intuition of this rule can be explained using Fig. 6. Another heuristic rule used in our algorithm is to give a higher preference to a fanin node with a smaller number of fanouts $n_o(x)$ because a node can be removed from the network if it has no fanouts after merging. The smaller the fanout number is,

the less duplication it will incur 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. The values of α, β, and γ in our experiment are 3, 0.5, and 0.5, respectively. We tried some experiments in which values are better for these parameters. However, there is no obvious winner. The node size bound is only for the runtime/quality tradeoff. The larger the node size is, the greater the runtime becomes. According to Theorem 1, the runtime for synthesizing one BDD is proportional to the square of the BDD node size. We tried node sizes in steps of 50 and found out that 200 is a reasonable limit for runtime purposes.

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
  $done \leftarrow 0$;
  repeat
    for each 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**
        $done \leftarrow 1$;
      while pq is not empty **do**
        $(g, in, out) \leftarrow pq$.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
      unmark all nodes;
  until $done = 1$
end

B. Dynamic Programming Algorithm to Synthesize One BDD for Delay Optimization

In this section, we synthesize a supernode of the collapsed network. Our algorithm uses a BDD to represent this node and recursively decomposes the BDD to form a Boolean network. During the decomposition, our algorithm tries to optimize the network mapping depth. The algorithm produces both the decomposed (synthesized) network and its mapping depth. Let us look at an example first. For the BDD in Fig. 4, there are five possible cuts, and each cut produces a different synthesis result. In our algorithm, we try all five cuts and choose the one producing the smallest mapping depth. Fig. 5(b) shows the decomposition produced by cut2 in Fig. 4. Obviously, we need to know the mapping depths of sub-BDDs before we can calculate the mapping depth of the BDD F in terms of cut2.

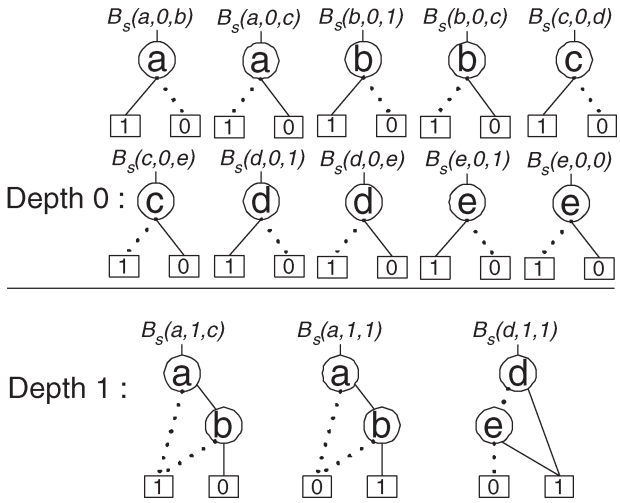


Fig. 7. All depth-0 sub-BDDs of the BDD in Fig. 5(a) and three depth-1 sub-BDDs.

To get the mapping depths of sub-BDDs, we recursively apply a dynamic programming algorithm. Our algorithm starts with the smallest sub-BDDs and processes sub-BDDs in increasing order of their depths.

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 [18]. After reordering, the algorithm processes sub-BDDs in increasing order of their depths, from depth 0 to depth $n-1$. For each depth l, the algorithm visits all BDD nodes. For each BDD node u, the depth of sub-BDD(u) is $n - l(u)$, its maximum possible cut level is $n - l(u) - 1$, and the algorithm generates the cut set $CS(u, l)$ 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 mapping depth for each cut (by calling the function $delayDecompose$), chooses the smallest mapping depth as the mapping depth of $\mathcal{B}_s(u, l, v)$, and saves the decomposition that produces the smallest mapping depth for this sub-BDD. The algorithm ignores those cuts with a large number of cut nodes (larger than $thresh$, which is 15 in our experiments) because large cuts generally do not produce good decompositions. Fig. 7 enumerates all depth-0 sub-BDDs of the BDD in Fig. 5(a) and three 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. 5(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. At the moment the algorithm starts to process BDD F in Fig. 4, the sub-BDDs that BDD F is decomposed into at every cut have already been processed, and we are able to choose the best cut for decomposing F and the corresponding mapping depth. 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 mapping depth of the synthesized network, and the synthesized network can be constructed by recursively applying saved best decompositions for $\mathcal{B}_s(r, n-1, 1)$ and sub-BDDs it decomposed into.

Algorithm 3: Logic synthesis algorithm for one BDD
Input: $inputDelay : inputDelay(x)$ is the mapping depth of the input variable x
Output: Synthesized network and its mapping depth
Local: $tmpDelay$, $bestDelay$: temporary variables
Global: $thresh$: a number to prune away large cuts
$delay : delay(\mathcal{B}_s(u,l,v))$ is the mapping depth 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;
for $l = 0$ **to** $n-1$ **do**
for each $u \in \mathcal{N}$ **do**
if $l(u) + l > n-1$ **then**
jump to the next iteration;
$enumerateCS(u,l)$;
for each $v \in CS(u,l)$ **do**
$bestDelay \leftarrow +\infty$;
if $l = 0$ **then**
$bestDelay \leftarrow inputDelay(V(u))$;
for $j = 0$ to $l-1$ **do**
if $|CS(u,j)| > thresh$ then
jump to the next iteration;
$tmpDelay \leftarrow delayDecompose(u,l,v,j)$;
if $tmpDelay < bestDelay$ **then**
$bestDelay \leftarrow tmpDelay$;
end
$delay(\mathcal{B}_s(u,l,v)) \leftarrow bestDelay$;
save the best decomposition for $\mathcal{B}_s(u,l,v)$;
end
end
end
produce the synthesized network by applying best decompositions for $\mathcal{B}_s(r,n-1,1)$ and its related sub-BDDs and set the network mapping depth to be $delay(\mathcal{B}_s(r,n-1,1))$, where r is the root of the BDD;
end

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

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
$CS(u,l) \leftarrow \varnothing$;
for each $v \in CS(u,l-1)$ **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

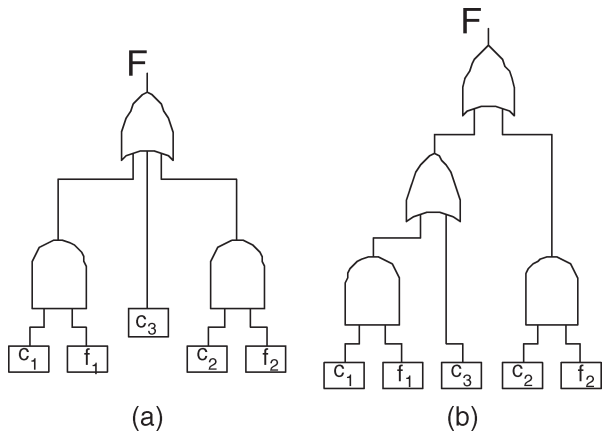


Fig. 8. Boolean network generated according to linear expansion at cut2 for the BDD in Fig. 4. Nodes c_1, f_1, c_2, f_2, and c_3 represent corresponding Boolean networks of sub-BDDs. (a) Network right after the linear decomposition. (b) One network after the large fanin OR gate is decomposed.

2) Producing the Mapping Depth for a Sub-BDD: This section illustrates how the function $delayDecompose$ (Algorithm 5) works using linear expansion. The function $delayDecompose$ first generates a set of AND gates with known mapping depths from the linear decomposition [see Fig. 8(a)], which are all fanins of an OR gate, and then, it uses a bin-packing-based algorithm to decompose the OR gate [see Fig. 8(b)] and produces the mapping depth of the decomposition.

Algorithm 5: $delayDecompose$
Input: u, l, v : specify sub-BDD $\mathcal{B}_s(u,l,v)$
j : the cut level
Output: the mapping depth of the linear decomposition at cut level j for sub-BDD $\mathcal{B}_s(u,l,v)$
begin
produce a set of AND gates using linear expansion for sub-BDD $\mathcal{B}_s(u,l,v)$ at cut j;
group these AND gates according to their mapping depths;
sort groups in increasing order of mapping depths;
for each *group in order* **do**
$d \leftarrow$ mapping depth of the group;
for each *gate in the group* **do**
$x \leftarrow$ number of inputs of the gate;
create a box with size x;
end
solve the bin-packing problem;
if *there is only one bin and the current group has the largest mapping depth* **then**
jump out of the loop;

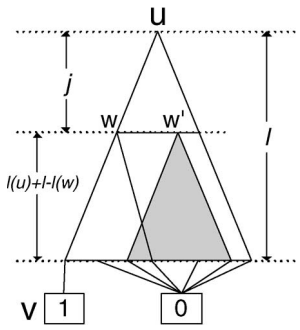


Fig. 9. Getting AND gates for $\mathcal{B}_s(u, l, v)$ using linear expansion at cut j.

end
for each *bin* **do**
create an OR gate with gates in the bin as inputs;
create a buffer and put it into the group with mapping depth $d + 1$;
end
end
$delay \leftarrow$ the mapping depth of the group with largest mapping depth plus one;
return $delay$;
end

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 Figs. 5(b) and 9], and the input mapping depth of the AND gate is the maximum mapping depth of its two inputs. Since we process sub-BDDs in increasing order of their depths, the mapping depths of both $\mathcal{B}_s(u, j, w)$ and $\mathcal{B}_s(w, l(u) + l - l(w), v)$ are known at this time. 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 equal to logic true. There are two cases in which a cut node $w' \in CS(u, j)$ does not have a corresponding AND gate in the decomposition.

1) The first case happens when $l(w') > l(u) + l$ and $w' \neq v$; in this case, w' is also a cut node of $CS(u, l)$, which means that w' is converted into terminal node 0 in $\mathcal{B}_s(u, l, v)$ (see Fig. 10).
2) The second case happens when $v \notin CS(w', l(u) + l - l(w'))$, as shown in Fig. 9; in this case, the sub-BDD starting from w' (the shaded area) is collapsed into logic 0, so we do not need to consider it.

In a linear-expansion-based decomposition, all the AND gates fan out to an OR gate, and our target is to decompose this large-input OR gate into K-input OR gates and then map all the gates to cells implementable by K-LUTs to achieve a small mapping depth. This problem can be formulated as a bin-packing decomposition problem [21], [22]. In general, the goal of bin packing is to find the minimum number of bins into which a set of boxes can be packed. In this paper, each box represents an AND gate, and the size of the box is the number of inputs to the AND gate. We use an algorithm similar to [22]. This algorithm is able to produce the optimum mapping depth

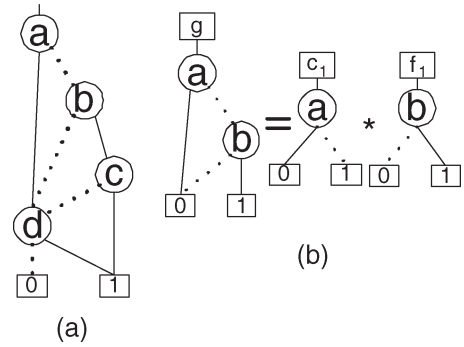


Fig. 10. Cut node without a corresponding AND gate in the decomposition. In (a), $CS(a, 0) = \{b, d\}$, $CS(a, 1) = \{c, d\}$, and $\mathcal{B}_s(a, 1, c)$ is the BDD g in (b). Let us decompose g using the cut at level 0 in (b). Since $d \in CS(a, 0)$ and $l(d) > l(a) + 0$, the cut node d is replaced to terminal node 0 in sub-BDD g, and it has no corresponding AND gate in the decomposition. Therefore, the sub-BDD is decomposed according to node b, the other node in $CS(a, 0)$. (a) BDD. (b) Cut node without a corresponding AND term in linear decomposition.

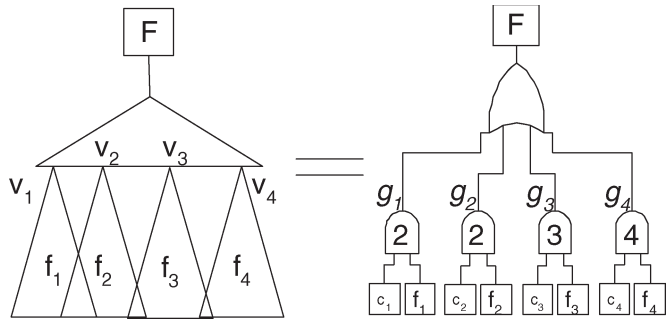


Fig. 11. BDD in this example is decomposed into an OR gate with four AND gates as inputs according to linear expansion. The numbers inside gates are mapping depths.

for $K \leq 6$ [22]. For simplicity, we assume that each buffer is an AND gate with only one input. First, the AND gates are grouped according to their mapping depths, with the gates having the same mapping depth belonging to the same group. Then, the algorithm processes the groups in increasing order of their mapping depths; for each group, the algorithm processes the gates in decreasing order of their input sizes and packs each gate into the first bin (the size of a bin is K) that has enough room. Each bin in a processed group with mapping depth d becomes a gate of the group with mapping depth $d + 1$, and the input size of this new gate is one. The algorithm terminates when there is only one bin left in the group with the highest mapping depth, and the mapping depth of the sub-BDD is the mapping depth of this group plus one.

Let us illustrate how an OR gate with many AND gates as inputs is decomposed using the bin-packing algorithm. Fig. 11 shows the linear decomposition result for a BDD at a cut with four cut nodes. According to linear expansion, the OR gate has four AND gates (g_1, g_2, g_3, and g_4) as inputs, and each AND gate is a two-input gate. The mapping depths of gates g_1 and g_2 are both 2, the mapping depth of gate g_3 is 3, and the mapping depth of gate g_4 is 4. In this example, let us assume that $K = 4$. In our algorithm, we first group these gates according to their mapping depths. Gates g_1 and g_2 belong to the group with mapping depth 2, gate g_3 belongs to the group with mapping depth 3, and gate

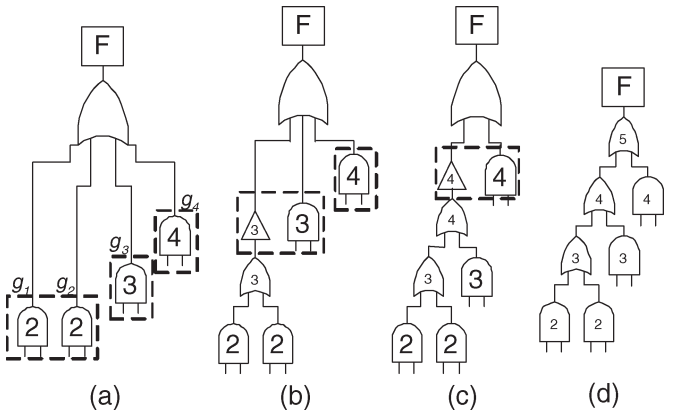


Fig. 12. Using bin packing to decompose an OR gate (step by step). The numbers inside the gates are gate mapping depths. (a) Creation of three groups for four gates according to mapping depths. (b) After processing the group with mapping depth 2. (c) After processing the group with mapping depth 3. (d) After processing the group with mapping depth 4.

g_4 belongs to the group with mapping depth 4 [see Fig. 12(a)]. We start by processing the group with mapping depth 2. For this group, there are two boxes, both with size 2, and they can be packed into one bin. Therefore, we create one bin for this group. Then, we create a buffer with mapping depth 3 and put it into the group with mapping depth 3 [see Fig. 12(b)]. As we can see in Fig. 12(b), we also create an OR gate for each bin, and the inputs of the OR gate are the gates inside the bin. After the group with mapping depth 2, we process the group with mapping depth 3, which has two gates, one having two inputs and the other having only one input. One bin is used to cover this group, and we create another buffer with mapping depth 4 and put it into the group with mapping depth 4 [see Fig. 12(c)], and so on and so forth. After we processed the group with mapping depth 4, our algorithm successfully decomposed the OR gate, as shown in Fig. 12(d). Finally, the mapping depth produced by the linear expansion for the BDD in Fig. 11 at the specified cut is 5.

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 linear expansion. We prefer special decompositions to linear expansion for these cases because these special decompositions use less sub-BDDs during decomposition. OR decomposition uses two sub-BDDs instead of the three used by linear expansion; MUX decomposition uses three sub-BDDs instead of four; and XNOR decomposition uses two sub-BDDs instead of four. 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 follows.

1) AND *decomposition*: This is a special case of linear expansion, where there is only one AND gate.
2) OR *decomposition*[2]: The condition for an OR decomposition is that $|CS(u,j)|=2$ and $v \in CS(u,j)$. Let us assume that $CS(u,j)=\{v,w\}$. In this case, $\mathcal{B}_s(u,l,v)$ can be decomposed as $\mathcal{B}_s(u,j,v) \vee \mathcal{B}_s(w,l(u)+l-l(w),v))$ because w is a 0-dominator of $\mathcal{B}_s(u,l,v)$. In Fig. 13(a), $\mathcal{B}_s(a,1,c)$ is a sub-BDD of the BDD in Fig. 5(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 $\mathcal{B}_s(a,0,c) \vee \mathcal{B}_s(b,0,c)$.
3) *MUX decomposition*: The condition for a MUX decomposition is $|CS(u,j)|=2$. Let $CS(u,j)=\{w_1,w_2\}$. In this case, $\mathcal{B}_s(u,l,v)$ can be decomposed as $(\mathcal{B}_s(u,j,w_1) \wedge \mathcal{B}_s(w_1,l(u)+l-l(w_1),v)) \vee (\neg\mathcal{B}_s(u,j,w_1) \wedge \mathcal{B}_s(w_2,l(u)+l-l(w_2),v))$. In Fig. 5, $CS(a,0)=\{b,c\}$, so the sub-BDD $\mathcal{B}_s(a,2,e)$ has a MUX decomposition at cut 0, and it is decomposed as $(\mathcal{B}_s(a,0,c) \wedge \mathcal{B}_s(c,0,e)) \vee (\neg\mathcal{B}_s(a,0,c) \wedge \mathcal{B}_s(b,1,e))$.
4) XNOR *decomposition*: The condition for an XNOR decomposition is that $|CS(u,j)|=2$ and the Boolean functions of w_1 and w_2 are complementary to each other, where $CS(u,j)=\{w_1,w_2\}$. In this case, $\mathcal{B}_s(u,l,v)$ can be decomposed as $\mathcal{B}_s(u,j,w_1) \overline{\oplus} \mathcal{B}_s(w_1,l(u)+l-l(w_1),v))$. XNOR decomposition is a special case of MUX decomposition.

[2]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.

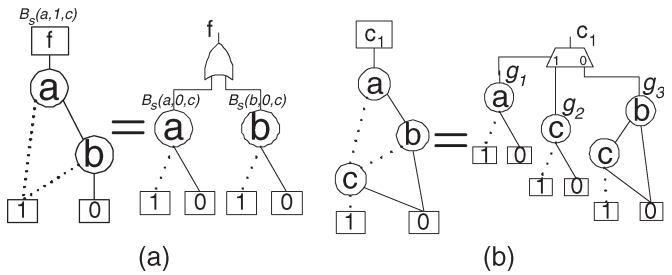


Fig. 13. Special decompositions. For the BDD in Fig. 4, (a) the OR decomposition can be applied to sub-BDD $\mathcal{B}_s(a,1,c)$ at cut 0, and $\mathcal{B}_s(a,1,c)=\mathcal{B}_s(a,0,c) \vee \mathcal{B}_s(b,0,c)$, and (b) the MUX decomposition can be applied to sub-BDD $c_1=\mathcal{B}_s(a,2,e)$ at cut 0, and $c_1=(g_1 \wedge g_2) \vee (\overline{g_1} \wedge g_3)$, where $g_1=\mathcal{B}_s(a,0,c)$, $g_2=\mathcal{B}_s(c,0,e)$, and $g_3=\mathcal{B}_s(b,1,e)$.

C. Complexity of the Algorithm

Let n denote the number of input variables and N denote the size of the BDD. The runtime and memory space of our algorithm is shown 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 still very 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 $O(n^2N^2)$, and the algorithm uses $O(nN^2)$ memory space.

Proof: Since each sub-BDD is determined by a root node, a cut level, and a node that corresponds to terminal node 1 of the sub-BDD (see Definition 7), the total number of sub-BDDs is limited by nN^2. As a result, the memory space needed by our algorithm is $O(nN^2)$. In the algorithm, we prune away the large cuts (cut size larger than $thresh$, which is 15 in our experiments), so the function $delayDecompose$ takes const time. Furthermore, the number of possible cuts for a sub-BDD is at most n. Therefore, the runtime of our algorithm is $O(n^2N^2)$.■

TABLE I
IN THIS TABLE, THE ROWS $Delay_{\rm w}$ ARE THE CIRCUIT MAPPING DEPTHS PRODUCED BY DDBDD WITH COLLAPSING, AND THE ROWS $Delay_{\rm wo}$ ARE THE CIRCUIT MAPPING DEPTHS PRODUCED BY DDBDD WITHOUT COLLAPSING

Ckt.	5xp1	9symml	apex6	apex7	c5315	clip
$Delay_w$	2	3	4	3	6	3
$Delay_{wo}$	3	5	5	4	8	4
Ckt.	count	des	rd84	rot	t481	vg2
$Delay_w$	3	4	3	6	2	4
$Delay_{wo}$	5	5	6	8	7	4

TABLE II
MAPPING DEPTHS REDUCED BY OUR BDD DECOMPOSITION ALGORITHM COMPARED TO BDS-PGA ON 103 COLLAPSED NODES

$\text{Delay}_{reduced}$	0	1	2	3	4	5
Number_{nodes}	4	69	14	10	5	1

IV. EXPERIMENTAL RESULTS

Our experiments are carried out on a desktop personal computer with a 3-GHz Intel Xeon central processing unit. The operating system is Red Hat Linux 8.0, and we use g++ to compile our programs. In the experiments, we use the circuit benchmarks provided by the authors of BDS-pga [12], which are used in [12] as well. The correctness of the synthesized circuits by our algorithm is verified using SIS. In this section, the first two experiments show the effectiveness of our collapsing algorithm and the BDD decomposition algorithm, and the last two experiments show the comparison of DDBDD with BDS-pga, SIS + DAOmap [6], and ABC [7]. In our experiments, we assume that the input size of an LUT is five, and the mapping depth is the number of LUT levels in the final mapped circuit.

To illustrate the effectiveness of our collapsing algorithm for mapping depth optimization, we compare the circuit mapping depths produced without the collapsing step and the mapping depths with the collapsing step. According to the experiments, our algorithm with collapsing always produces circuits with better or equal mapping depths. The collapsing is very effective mainly due to the following reasons: 1) It can remove some logic redundancies such as those caused by local reconvergence, and 2) it makes an individual node larger than before, thus producing more opportunities to find better decompositions. Table I shows the mapping depth comparison results on some of the circuits.

To illustrate the effectiveness of our dynamic-programming-based BDD decomposition algorithm, we perform the following experiment. We first run our collapsing algorithm on all the benchmark circuits and choose the collapsed nodes with a BDD size larger than 50, and then, we run both our BDD decomposition algorithm and the BDD decomposition algorithm in BDS-pga on these nodes and compare the mapping depths. There are 103 such nodes, and our decomposition algorithm performs uniformly better than the decomposition algorithm of BDS-pga in terms of mapping depth. Table II shows the statistics. For all 103 nodes, our algorithm reduces the mapping depths by one level for 69 nodes, by two levels for 14 nodes, by three levels for ten nodes, by four levels for five nodes, and by five levels for one node. The sum of mapping depths produced by our algorithm is 292 for all these nodes, and the sum of mapping depths by BDS-pga is 444. These results show that our BDD decomposition technique is effective for mapping delay minimization.

TABLE III
COMPARISON RESULTS OF OUR ALGORITHM WITH BDS-PGA, SIS + DAOMAP, AND ABC. IN THIS TABLE, COLUMNS "DELAY" REPRESENT DEPTHS OF THE MAPPED CIRCUITS, AND COLUMNS "AREA" REPRESENT THE NUMBER OF LUTS OF THE MAPPED CIRCUITS

	DDBDD		BDS-pga		SIS+DAOmap		ABC	
Ckt.	Delay	Area	Delay	Area	Delay	Area	Delay	Area
5xp1	2	19	2	16	3	26	3	20
9sym	3	8	3	8	5	63	5	57
9symml	3	8	3	8	5	70	4	54
alu2	4	70	5	45	7	121	6	123
apex6	4	249	6	194	4	188	4	168
apex7	3	113	5	69	4	61	4	57
b9	3	55	3	43	3	40	2	38
c1355	4	72	4	66	4	68	4	71
c1908	7	228	9	118	8	173	7	99
c499	4	71	4	65	4	68	4	77
c5315	6	553	8	445	8	365	7	361
c880	7	193	9	123	8	147	6	100
clip	3	32	5	42	6	48	4	75
count	3	50	5	34	3	41	3	35
des	4	1098	4	916	7	1045	5	964
duke2	3	208	7	180	5	170	5	169
misex1	2	14	2	14	2	16	2	15
rd84	3	16	3	14	5	95	5	182
rot	6	301	10	223	6	217	6	207
t481	2	5	2	5	6	121	6	247
vg2	4	80	5	61	3	30	4	43
Total	80	3443	104	2689	106	3173	96	3162
Norm	1	1	1.3	0.78	1.33	0.92	1.2	0.92

Table III shows the comparison of our algorithm with BDS-pga, SIS + DAOmap, and ABC. In the table, the data of BDS-pga are from [23]. BDS-pga employs the MFFC-based collapsing algorithm, which is followed by a heuristic BDD decomposition method. For SIS, the benchmark circuits are first optimized using the scripts *script.rugged* and *script.delay*[3]; the optimized circuits are then decomposed into circuits consisting only of two-input gates by the commands *tech_decomp -a 1000 -o 1000* and *dmig -k 2*; after the decomposition, the command *daomap -k 5* is used to cover the circuits by LUTs, and the results are reported. For ABC, we run five passes of commands *choice;fpga* before the command *ps*. In this table, the last row, i.e., *Norm*, is for the normalized values of the other three methods compared to DDBDD. The average runtime of DDBDD is less than 1 min per circuit. Compared to our algorithm, on the average, BDS-pga produces circuits with 30% more mapping depth and 22% less area, SIS + DAOmap produces circuits with 33% more mapping depth and 8% less area, and ABC produces circuits with 20% more mapping depth and 8% less area.

To illustrate the scalability of our algorithm, we perform a comparison of DDBDD and BDS-pga on the ten largest Microelectronics Center of North Carolina (MCNC) combinatorial benchmarks. Table IV shows the results. The circuits are first mapped by both DDBDD and BDS-pga. To see the impact of mapping depth reduction, we then feed the circuits into VPR [24] to run placement and routing. We use an LUT of size five, a cluster of size ten, and length-4 wire segments in the experiment. VPR is run in the timing-driven mode. We use a 100-nm technology node, which is the same as that used in [25]. We first run VPR to obtain routing results with the minimum number of routing tracks for each circuit and then apply additional 20%

[3]We only apply script *script.rugged* to circuit *C5315* because the script *script.delay* fails to run on it.

TABLE IV
COLUMNS DEPTH SHOWS THE MAXIMUM DEPTH OF THE MAPPED CIRCUIT, COLUMNS LUTS SHOW THE NUMBER OF LUTS, COLUMNS DELAY SHOW THE MAXIMUM DELAY ACCORDING TO VPR, AND COLUMNS RUNTIME ARE THE RUNTIME OF THE PROGRAM. UNDER THE COLUMN 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	47.7	11	1116	1.32E-08	5.2	1.57	0.90	1.26
apex2	8	1604	1.28E-08	101.4	14	1317	1.49E-08	4.5	1.75	0.82	1.17
apex4	6	1314	1.03E-08	100.9	15	991	1.33E-08	5.5	2.5	0.75	1.29
des	6	1552	1.24E-08	49.9	7	1106	1.18E-08	4.6	1.17	0.71	0.95
ex1010	7	4456	1.55E-08	548.4	20	3636	2.25E-08	1078	2.86	0.82	1.46
ex5p	6	1381	1.04E-08	169.6	11	810	1.22E-08	2.6	1.83	0.59	1.17
misex3	6	1224	1.03E-08	50.5	10	1032	1.13E-08	4.2	1.67	0.84	1.09
pdc	8	4042	1.57E-08	379.8	18	3210	2.17E-08	31	2.25	0.79	1.38
seq	6	1518	9.59E-09	79.5	12	1214	1.29E-08	6.3	2	0.80	1.34
spla	8	3243	1.54E-08	1483.9	15	2659	2.12E-08	25.3	1.88	0.82	1.37
average				301.2				116.7	1.95	0.78	1.25
Geometric mean									1.89	0.78	1.24

routing tracks and rerun routing to get the final results, as commonly practiced [24]. To have a fair comparison, for each circuit, we apply the same routing track count for both DDBDD and BDS-pga (we pick the smaller track count required between DDBDD and BDS-pga). For example, if it was determined that a circuit had a minimum track count of 10, then both BDS-pga and DDBDD will be run on an FPGA with a track count of 12 (i.e., $10 + 20\%$). The maximum delay of each circuit is collected. Table IV shows that on the average, BDS-pga produces circuits with 95% more mapping depth (or 25% more delay after placement and routing) with 22% less area. Although the runtime of DDBDD is still reasonable, it is several times larger than BDS-pga mainly due to the collapsing procedure, which runs for many iterations. We will reduce the runtime in our future work.

We would like to mention that DDBDD does not work well for datapath-intensive circuits. The largest MCNC benchmark circuits are mainly such type of circuits. On these circuits, the performance of DDBDD is not good compared to DAOmap and the ABC mapper. For example, DDBDD maps those circuits in Table IV with 8% more mapping depths and 34% more area compared to DAOmap on the average. However, we believe that datapath circuits are not the right circuits to evaluate the quality of an FPGA technology mapping algorithm. Modern FPGA chips (such as Xilinx and Altera FPGA chips) have dedicated logic blocks for datapath elements, such as multipliers. Given an FPGA design, a commercial software first infers datapath logic elements from the design. After inference, most of the datapath elements are mapped, using a template-based method, onto dedicated logic blocks. Then, the rest of the circuit, mostly random logic, is mapped using a technology mapping algorithm. Therefore, it is more accurate to use random logic circuits to verify the quality of an FPGA technology mapping algorithm.

Normally, datapath circuits are all well designed with good and regular structures. BDD-based technology mapping algorithms build BDDs for the entire network and then decompose BDDs to form a new network. Therefore, BDD-based technology mapping algorithms tend to destroy the original circuit structure. As a result, BDD-based algorithms are usually not performing well for datapath circuits. For random logic circuits, there are no fixed best circuit structures. BDD-based methods can find good structures using a BDD size reduction algorithm by reordering the variable orders. Based on these analyses, we further compare our algorithm with BDS-pga, SIS + DAOmap, and ABC on nine control circuits from the MCNC benchmark [26]. Table V shows the comparison results. The average runtime of our algorithm on these circuits is less than 1 s per circuit. In the table, we again observe that DDBDD outperforms other algorithms on mapping depth. In general, we show that our solution provides a significant amount of performance gain, which makes the tradeoff on area worthwhile if the design goal is for high performance.

TABLE V
COMPARISON RESULTS OF OUR ALGORITHM WITH BDS-PGA, SIS + DAOMAP, AND ABC ON NINE CONTROL CIRCUITS

Ckt.	DDBDD		BDS-pga		SIS+DAOmap		ABC	
	Delay	Area	Delay	Area	Delay	Area	Delay	Area
s1488	3	214	6	229	4	201	4	182
s382	3	55	3	45	3	40	3	38
s400	3	54	3	45	3	41	3	38
s510	3	73	4	66	4	81	4	73
s526n	3	50	3	46	3	59	3	33
s1494	3	225	6	221	4	197	4	187
s386	3	49	4	54	3	53	3	42
s444	3	58	3	50	3	43	3	39
s526	3	50	3	46	3	59	3	33
Total	27	828	35	802	30	774	30	665
Norm	1	1	1.30	0.97	1.11	0.93	1.11	0.80

V. CONCLUSION

In this paper, we have 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 the circuit mapping depth. BDD decomposition was mainly carried out through linear expansion and was further enhanced by special decomposition cases when necessary. Our algorithm was delay-centric overall. In particular, the dynamic programming approach was designed to efficiently search through all the possible decompositions in a BDD to achieve the minimal mapping depth among these decompositions. We showed that we were able to achieve a 30% performance gain with a 22% area overhead compared to previous state-of-the-art BDD-based logic synthesis algorithms for FPGAs. Our future work will consider area reduction techniques during BDD decomposition. We can also explore different variable reordering techniques based on

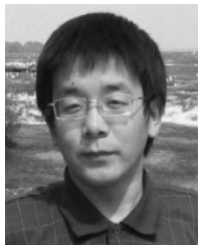
the timing criticality of BDD nodes so that noncritical BDD nodes can be optimized toward area reduction.

ACKNOWLEDGMENT

The authors would like to thank Prof. R. Tessier of the University of Massachusetts, Amherst, for his assistance in providing them with the BDS-pga program and the benchmark suites. The authors would like to thank Intel for the equipment used in the study.

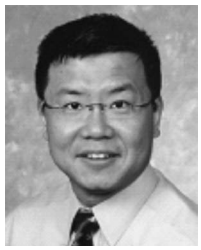
REFERENCES

[1] R. K. Brayton, G. D. Hachtel *et al.*, *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA: Kluwer, 1984.
[2] H. Savoj, R. K. Brayton, and H. Touati, "Extracting local don't cares for network optimization," in *Proc. ICCAD*, 1991, pp. 514–517.
[3] H. Savoj and R. K. Brayton, "The use of observability and external don't cares for the simplification of multi-level networks," in *Proc. DAC*, 1991, pp. 297–301.
[4] E. Sentovich, K. Singh *et al.*, "SIS: A system for sequential circuit synthesis," EECS Dept., Univ. California, Berkeley, CA, Tech. Rep. UCB/ERL Memorandum M89/49, 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," *IEEE Des. Test. Comput.*, vol. 9, no. 3, pp. 7–20, Sep. 1992.
[6] D. Chen and J. Cong, "DAOmap: A depth-optimal area optimization mapping algorithm," in *Proc. ICCAD*, 2004, pp. 752–759.
[7] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 240–253, Feb. 2007.
[8] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 1, pp. 1–12, Jan. 1994.
[9] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs," in *Proc. FPGA*, 2006, pp. 41–49.
[10] S. C. Chang, M. M. Sodowska, and T. Hwang, "Technology mapping for TLU FPGAs based on decomposition of binary decision diagrams," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 15, no. 10, pp. 1226–1236, Oct. 1996.
[11] C. Yang and M. Ciesielski, "BDS: A BDD-based logic optimization system," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 7, pp. 866–876, Jul. 2002.
[12] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," *ACM Trans. Design Autom. Electron. Syst.*, vol. 7, no. 4, pp. 501–525, Oct. 2002.
[13] V. Manohararajah, D. P. Singh, and S. D. Brown, "Post-placement BDD-based decomposition for FPGAs," in *Proc. FPL*, 2005, pp. 31–38.
[14] R. Brayton, R. Rudell *et al.*, "MIS: A multiple-level logic optimization system," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. CAD-6, no. 6, pp. 1062–1081, Nov. 1987.
[15] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.
[16] C. Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell Syst. Tech. J.*, vol. 38, no. 4, pp. 985–999, Jul. 1959.
[17] S. B. Akers, "Functional testing with binary decision diagrams," in *Proc. 8th Annu. Conf. Fault Tolerant Comput.*, 1978, pp. 82–92.
[18] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. ICCAD*, 1993, pp. 42–47.
[19] K. Karplus, "Using if-then-else DAGs for multi-level logic minimization," Univ. California, Santa Cruz, CA, Tech. Rep. UCSC-CRL-88-29, 1989.
[20] C. Yang, "BDD-based logic synthesis system," Ph.D. dissertation, EECS Dept., Univ. Massachusetts, Amherst, MA, 2000. Tech. Rep.
[21] R. J. Francis, J. Rose, and Z. G. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," in *Proc. DAC*, 1991, pp. 227–233.
[22] R. J. Francis, J. Rose, and Z. G. Vranesic, "Technology mapping lookup table-based FPGAs for performance," in *Proc. ICCAD*, 1991, pp. 568–571.
[23] [Online]. Available: http://www.ecs.umass.edu/ece/tessier/rcg/bds-pga-2.0/results_bds-pga.html
[24] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, MA: Kluwer, 1999.
[25] F. Li, D. Chen, L. He, and J. Cong, "Architecture evaluation for power-efficient FPGAs," in *Proc. FPGA*, 2003, pp. 175–184.
[26] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0.," Microelectron. Center North Carolina, Research Triangle Park, NC, 1991. Tech. Rep.



Lei Cheng received the B.S. degree in computer science from the University of Science and Technology of China, Hefei, China, in 1999, the M.S. degree in computer science from the Chinese Academy of Sciences, Beijing, China, in 2002, and the Ph.D. degree in computer science from the University of Illinois, Urbana–Champaign, in 2007.

He is currently with Synplicity, Inc., Sunnyvale, CA. His main research interest is computer-aided design for VLSI, primarily in the area of logic synthesis and physical synthesis for FPGAs.



Deming Chen received the B.S. degree from the University of Pittsburgh, Pittsburgh, PA, in 1995 and the Ph.D. degree from the University of California, Los Angeles, in 2005.

He was a Software Engineer from 1995 to 1999 and from 2001 to 2002. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Illinois, Urbana–Champaign (UIUC). He is also with the Coordinated Science Laboratory, UIUC. He has been actively contributing to various leading computer-aided design (CAD) conference proceedings and journals in the field of high-level and logic synthesis, low-power design, and FPGA design and synthesis. Some of his research ideas have already been incorporated in commercial software that is distributed to many customers of leading companies (e.g., Altera and Magma). His current research interests include CAD for FPGA, nanosystems design and nanocentric synthesis, microprocessor architecture design under process variation, and reconfigurable computing.

Dr. Chen received the Achievement Award for Excellent Teamwork from Aplus Design Technologies in 2001, the Arnold O. Beckman Research Award from UIUC in 2007, and the National Science Foundation CAREER Award in 2008. He serves as a Technical Committee Member and Session Chair in a series of conferences and symposia.

Martin D. F. Wong (M'88–SM'04–F'06) received the Ph.D. degree in computer science from the University of Illinois, Urbana–Champaign (UIUC), in 1987.

He is currently a Professor with the Department of Electrical and Computer Engineering, UIUC. He is also with the Coordinated Science Laboratory, UIUC. Before he joined UIUC, he was a Bruton Centennial Professor of computer sciences with the University of Texas, Austin. He is the author of more than 300 published technical papers and has graduated 35 Ph.D. students. His research interests include computer-aided design (CAD) of very-large-scale integration (VLSI) circuits, design and analysis of algorithms, and combinatorial optimization.

Dr. Wong was an IEEE Distinguished Lecturer from 2005 to 2006. He has served as the Technical Program Chair, the General Chair, and a Steering Committee Member for the Annual ACM International Symposium on Physical Design. He has also served on the Technical Program Committees of numerous VLSI/CAD conferences. He has served as an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN (2002–2005) and the IEEE TRANSACTIONS ON COMPUTERS (1995–2000). He has also served as a Guest Editor of four Special Issues of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN. He is the recipient of the 2000 IEEE CAD TRANSACTIONS Best Paper Award for his work on interconnect optimization. He also received Best Paper Awards at the 1986 ACM/IEEE Design Automation Conference (DAC) and 1995 International Conference on Computer Design for his work on floorplan design and routing, respectively. His paper on circuit partitioning at the 1994 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) has been included in the book *The Best of ICCAD—20 Years of Excellence in Computer Aided Design*. He is currently on the Editorial Board of the *ACM Transactions on Design Automation of Electronic Systems*.