BDD-based Logic Synthesis for LUT-based FPGAs

Navin Vemuri
Intel Corporation, Hillsboro, OR
and
Priyank Kalla
University of Utah, Salt Lake City, UT
and
Russell Tessier
University of Massachusetts, Amherst, MA

Contemporary FPGA synthesis is a multi-phase process which involves technology independent logic optimization followed by FPGA-specific mapping to a target FPGA technology. Conventional technology-independent transformations target standard cells and are unable to optimize circuits with constraints and goals specific to FPGA architectures. This paper describes an *FPGA-specific* logic synthesis approach, which unites multi-level logic transformation, decomposition, and optimization techniques into a single synthesis framework. This system performs network transformation, decomposition and optimization at an early stage to generate a network which can be *directly* mapped onto FPGAs. Our techniques are built upon a BDD-based logic decomposition system. With this system, both AND-OR decompositions *and* AND-XOR decompositions can be identified, resulting in large area savings for synthesized XOR-intensive circuits.

To induce good decompositions, a *maximum fanout free cone* (MFFC) based partial clustering and collapsing technique is used. This step is followed by an area-minimizing variable partitioning heuristic which decomposes collapsed nodes into LUT-feasible sub-functions. As a post-processing step, a performance-driven re-synthesis phase is performed to alleviate increased delay caused by excessive logic sharing. We compare the quality of results obtained using our techniques with those of academic (BoolMap, SIS) and industry (Altera Quartus) FPGA synthesis tools. Experimental results indicate that the circuits generated by our techniques are not only smaller, but are also significantly faster than those synthesized by conventional FPGA synthesis tools. Furthermore, the computation times required by our techniques are significantly smaller than those of previous techniques.

Categories and Subject Descriptors: B.6.3 [**Hardware**]: Logic Design

General Terms: Algorithms, Design

Additional Key Words and Phrases: FPGA, logic synthesis, BDD, decomposition

---

This work was performed at the University of Massachusetts, Amherst. P. Kalla and R. Tessier were supported by National Science Foundation grants CCR-0081405 and CCR-9901254.
Author's address: R. Tessier, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003; email: tessier@ecs.umass.edu.

1. INTRODUCTION

In this paper, a logic synthesis approach for LUT-based FPGA architectures is presented. This approach encompasses a complete FPGA-specific logic synthesis system, which includes network transformations, *technology-dependent* logic decomposition, optimization and technology mapping. Earlier FPGA synthesis research [Murgai et al. 1990] [Babba and Crastes 1992] advocated technology-independent logic synthesis followed by technology mapping onto FPGA architectures. This approach generates minimized multi-level logic nodes and maps nodes to LUTs as a *post-processing* step [Murgai et al. 1995]. A number of FPGA technology mapping approaches have been proposed in the literature [Murgai et al. 1990][Francis et al. 1990] [Francis et al. 1995] [Filo et al. 1991] [Sawkar and Thomas 1992] [Cong and Ding 1994]. Although many techniques demonstrate robustness [Chen et al. 1992] and optimality [Cong and Ding 1994] in mapping circuits to FPGAs, they are used subsequent to, and in isolation with, logic optimization. The resulting area/delay characteristics of the circuits may not be satisfactory, especially for large designs.

Following these earlier efforts, an attempt was made to restructure/optimize logic to facilitate mapping onto FPGA architectures. Karplus [Karplus 1991] proposed Xmap, which uses an **if-then-else** (ITE) DAG to represent functions and uses cofactoring for functional decomposition. Subsequent bin-packing techniques are used for technology mapping. Recent techniques [Legl et al. 1996] [Eckl et al. 1996][Jiang et al. 1997] [Stanion and Sechen 1995] follow a three step approach: First, circuit optimization using conventional technology independent optimization tools such as SIS [Sentovich et al. 1992] is performed. Architecture specific network transformation heuristics are then used to reduce the logic depth and create k-feasible *supernodes*, where k is the input count of each LUT. Finally, mapping algorithms are used to realize the circuits in the desired FPGA architecture.

A common feature of the above techniques is the initial technology independent optimization step. The implications of performing conventional technology independent optimization, with tools such as SIS, prior to and in isolation with FPGA-specific restructuring and subsequent mapping are as follows:

(1) Conventional synthesis tools perform a variety of network transformations and optimizations which are geared specifically towards *standard cell* architectures. Knowledge of FPGA architectures is not used in the optimization process. The goal of standard cell-based, multi-level logic optimization is literal count minimization. For LUT-based FPGAs, resource consumption depends on the the number of look-up table inputs. When mapped to LUTs, a literal-minimized network may lead to a suboptimal implementation.

(2) Larger nodes need to be decomposed into smaller, less complex, subfunctions. Logic optimization techniques rely extensively on algebraic factoring and decomposition. Conventional AND-OR intensive factoring heuristics are unable to efficiently generate subfunctions that can be directly mapped onto LUTs.

(3) Conventional logic synthesis tools resort to aggressive factoring (kernel extraction) to reduce the number of literals, often resulting in high-fanout nodes. Although this may be acceptable for standard cell implementations, FPGAs suffer from limited resources; often resulting in routing difficulties. Additionally, aggressive extraction often creates excess nodes, which results in resource

consumption and a degradation in area and delay characteristics.

Recently, [Chen and Cong 2001] reported that the impact of logic decomposition on delay and area cannot be forecast accurately. As a result, they suggest that mapping be performed simultaneously over a *set of decompositions* so that a single, best solution may be chosen. This approach combines *technology decomposition* with *technology mapping*. However, initial logic optimization is still conventional (e.g. based on SIS), rather than technology-specific. Our approach takes the target technology (k-input LUTs) into consideration during the technology-independent logic optimization and decomposition phase.

1.1 Research Contributions

To overcome the limitations of previous FPGA synthesis techniques, we have developed a LUT-based FPGA synthesis approach that unifies technology-independent logic optimization with LUT-based logic restructuring. We guide network transformation and subsequent logic decomposition and optimization to generate a network which can be efficiently mapped to LUT-based FPGAs. To facilitate transformations, our approach makes extensive use of binary decision diagrams (BDDs).

BDDs have been exploited by CAD engineers for FPGA synthesis [Legl et al. 1996] [Chang et al. 1996] [Sawada et al. 1995] [Lai et al. 1993] [Jiang et al. 1997], primarily in the logic decomposition phase. Recently, a generic BDD-based functional decomposition approach, **BDS** [Yang 2000], was presented which can efficiently identify both *algebraic* and *Boolean* (AND-OR and AND-XOR) decompositions. Using a functional decomposition engine similar to that of BDS, we present a comprehensive logic synthesis system to specifically target logic decompositions for k-feasible LUTs. Binary decision diagrams [Bryant 1986] are used to represent Boolean functions and to aid in the creation of logic decompositions. To induce good decompositions, a *maximum fanout free cone* (MFFC) based partial collapse technique is applied in the Boolean domain. Efficient BDD-based variable partitioning heuristics are subsequently used to decompose all collapsed nodes into k-LUT feasible subfunctions. These transformations are performed to achieve minimized LUT area. When combined, the FPGA-specific optimizations form the heart of our synthesis framework, **BDS-pga** (BDD-based decomposition system for FPGAs).

Our synthesis process is completed with **FlowMap** [Cong and Ding 1994], an academic technology mapping tool. FlowMap uses a max-flow, min-cut algorithm for depth optimal mapping of k-feasible nodes on k-input LUTs and incorporates heuristics to perform area-minimal mapping. The tool pre-processes the optimized network by performing a two-input AND-OR decomposition on each network node. Such restructuring can potentially undo the decompositions performed by BDS-pga. Since BDS-pga already creates a k-feasible network, this two-input AND-OR decomposition feature of FlowMap is disabled, so that BDS-pga's logic decomposition is not destroyed.

To reduce critical-path delay, a performance-driven re-synthesis step is incorporated into BDS-pga. Circuit delay can be reduced by minimizing the level count topologically using 1) controlled clustering and collapsing, 2) re-decomposition, 3) logic simplification and 4) technology re-mapping. Experiments using area minimization and subsequent delay re-synthesis indicate that circuits generated using our techniques are not only smaller, but are also significantly faster than those synthesized using conventional FPGA synthesis approaches.

The remainder of the paper describes our logic synthesis infrastructure. Section 2 introduces the terminology used in this paper and reviews logic decomposition using BDS. In Section 3, an MFFC-based iterative partial collapse approach is presented. Section 4 describes two decomposition algorithms which are based on variable partitioning of BDD graphs. Section 5 focuses on delay-reduction strategies. Experimental results are presented in Section 6 and the benefits and limitations of BDS-pga are analyzed. Section 7 discusses possible future work and concludes the paper.

2. PRELIMINARIES

A combinational Boolean network η can be represented by a directed acyclic graph $\eta = (V, E)$, where each node $v \in V$ represents an arbitrary logic gate and each directed edge $(u, v) \in E$ represents a connection from the output of the node u to the input of node v. The *depth* of a node v is the number of edges on the longest path from a primary input (PI) to v. In this paper, we define the depth of each PI as one. The depth of a network is the largest depth amongst all nodes in the network. Let $input(v)$ and $fanout(v)$ represent the set of fanins and the set of fanouts of node v, respectively. The *support* of a Boolean expression F is the set of variables, $\mathcal{S_F}$, that F explicitly depends on, *i.e.* the fanin of that node. For example, if $F = xy + x\prime y\prime$, $\mathcal{S_F} = \|(x, y)\| = 2$. A *binary decision diagram (BDD)* is a rooted, directed acyclic graph (DAG) representing a switching function [DeMicheli 1994] with an unconstrained number of in-edges and two out-edges, one each for the one and zero decision paths of a given variable.

A *fanin cone* C_v rooted at v is a connected subnetwork consisting of v and its predecessors. A *fanout free cone* (FFC) is a subnetwork where no node in the cone is connected to a node not in the cone. A *maximum fanout free cone* (MFFC) is the largest possible FFC in terms of node count. A *k-feasible* node has no more than k inputs. All nodes in a k-feasible network are k-feasible.

In a network η, the topologically longest path(s) are considered to be critical. Nodes on the critical path(s) are critical nodes. In this paper, we refer to ϵ-critical paths as those paths whose topological lengths are within ϵ of the longest path. $\mathcal{L}$ refers to the longest critical path, while $\mathcal{L}_\Upsilon$ refers to the set of all longest paths. $\mathcal{L}_\epsilon$ is the set of paths that are ϵ-critical.

2.1 Review of BDD-based Logic Optimization

In this subsection, basic BDD-based logic decomposition theory presented in [Yang 2000], is described in the context of our work. For a detailed description of the **BDS** decomposition system, the reader is referred to [Yang et al. 1999] [Yang 2000].

BDD-based Functional Decomposition: In our synthesis framework, BDDs [Bryant 1986] are the functional representation of choice. First, a Reduced Ordered Binary Decision Diagram (ROBDD) is built for a function. BDDs for certain classes of functions are *exponential* in the number of variables and cannot be constructed. To overcome this problem, *partitioned-ROBDDs* with intermediate variables [Narayan et al. 1996] are used to further partition the functions. Since each function partition (a subfunction in itself) is represented by a BDD, partitioned

Table 1. Dominators and their corresponding decompositions

Type	BDD Structure	Decomposition
1	*1-dominator*	algebraic AND
2	*0-dominator*	algebraic OR
3	*x-dominator*	algebraic XNOR
4	*generalized dominator*	Boolean AND/OR
5	*generalized x-dominator*	Boolean XNOR
6	*cof. wrt. single node*	simple MUX
7	*cof. wrt. supernode*	functional MUX

ROBDDs present an initial circuit partitioning for decompositions.

The theory of *dominators* [Yang et al. 1999] forms the basis for BDD-based decomposition. A BDD traversal (scan) is performed using a depth-first search to identify dominators, structural features that indicate the convergence of positive and negative BDD edges at a particular node. After dominator identification, a *cut* is performed on the BDD which divides the BDDs into two parts, creating a decomposition. A list of dominators and their corresponding decompositions appears in Table 1.

The decomposition engine performs a search for efficient BDD decompositions, from the most efficient (algebraic) to less efficient decompositions (Boolean). The engine first searches for a simple disjunctive (algebraic) decomposition. These decompositions are based on 1,0 and x-dominators, which are critical points on the BDD where simple decompositions (AND, OR and XOR decompositions, respectively) can be performed. A 1-dominator lies on all paths from the root node to the **constant 1** node. When disjunctive decomposition fails, the BDDs are decomposed using generalized dominators. As a last resort, the BDD is decomposed by cofactoring with respect to the top variable.

As shown in Fig. 1, the *bound set* corresponds to the *divisor* **D** during decomposition. The *free set* constitutes the variables below the cut. **D** is a generalized dominator since it does not necessarily lead to an algebraic (disjoint) decomposition. All dangling solid edges are tied to leaf node **1**, as shown in Fig. 1(b). The *quotient* of this division is obtained from **F** by setting the off-set of the divisor **D** to be a don't care, and performing don't care minimization. The *restrict* algorithm [Coudert and Madre 1990] is used for this purpose. **F** is then minimized with this don't care in Fig. 1(c). The minimized function is the quotient **Q** of the division. This decomposition is a case of non-disjoint **Boolean** decomposition.

Prior to decomposition, network transformation procedures such as *sweep*, *eliminate* and *re-substitution* are used as BDD operations. These operations are an essential part of logic synthesis systems and help in restructuring and optimizing the Boolean network.

2.2 A case for using BDD-based decomposition for FPGA synthesis

An important part of BDS is the decomposition engine that can identify both AND/OR and AND/XOR decompositions efficiently. XOR identification can significantly reduce implementation resources if efficient XOR implementations are

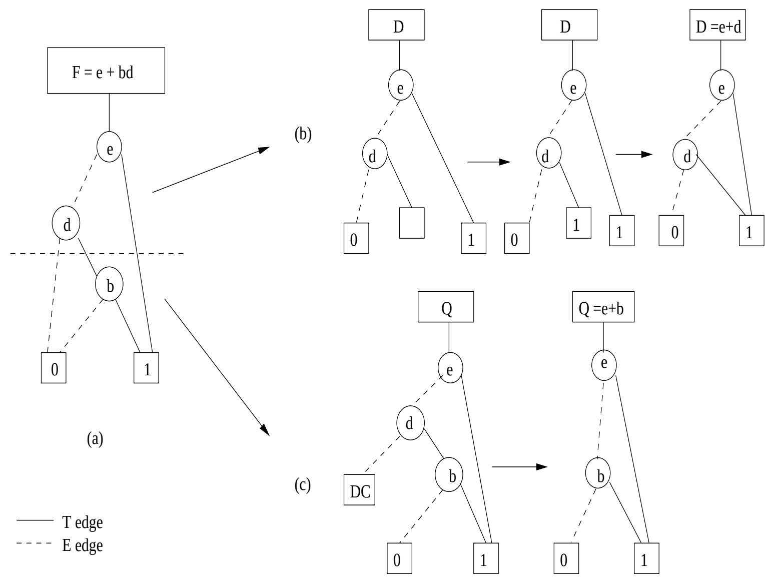

Fig. 1. A simple example of Boolean division.

Table 2. Preliminary Experiments: # of LUTs

Ckt.	sc.rug+FlowMap	BDS+FlowMap
9sym	135	8
C499	66	70
C880	110	139
alu2	153	94
alu4	236	193
apex6	235	200
b9	51	42
clip	103	62
des	1291	983
duke2	163	187
f51m	32	14
rot	302	259
t481	401	5

available. Unlike standard cell designs, where decomposition significantly depends upon the type (functionality) of cells present in the technology library, decomposition for LUT-based FPGAs depends upon the number of inputs to, and not the functionality of, the decomposed nodes. These features, combined with the speed of BDD-based decomposition (a by-product of simple graph traversal), makes it a natural choice for LUT-based FPGA synthesis.

Prior to creating new BDD-based algorithms, we carried out comparative experiments for SIS and BDS and produced optimized networks to be mapped onto FPGAs. For technology mapping, we used the RASP tool suite (FlowMap, FlowSyn) [Cong et al. 1996]. The LUT area of circuits synthesized using SIS (with script **script.rugged**) and FlowMap were compared against those generated by using BDS and FlowMap. Results in the last column of Table 2 indicate a 31% improve-

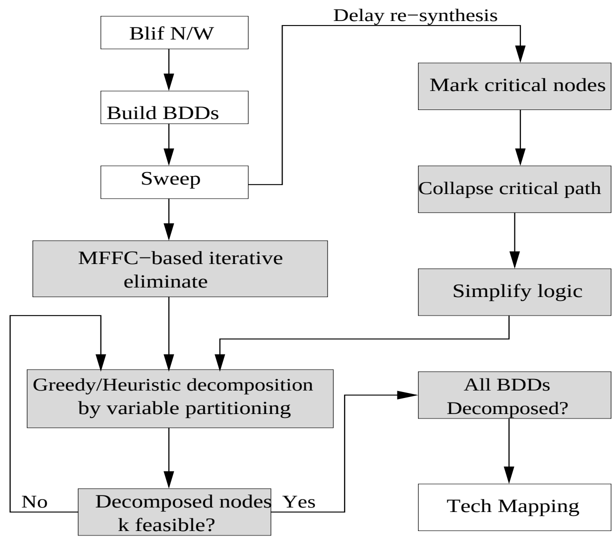

Fig. 2. The synthesis flow for BDS-pga.

ment in area for BDS and FlowMap over results obtained using SIS and FlowMap. The BDS and FlowMap circuits were on average 8% faster than those obtained using SIS and FlowMap.

Although BDS produces good results for FPGAs, it is desirable to optimize BDD-based functional decomposition specifically for k-input LUT-based FPGAs. The synthesis flow for **BDS-pga**, our BDD-based FPGA logic synthesis system, is shown in Fig. 2. Shaded blocks indicate the modifications and enhancements to the contemporary logic synthesis flow for FPGAs. Our new approach incorporates the following steps:

MFFC-based Eliminate. The eliminate procedure collapses the network within its maximum fanout-free cone to create a cluster of variable partition size. This step clusters and collapses nodes into supernodes to induce good decompositions.

Decomposition. Two decomposition schemes which generate area-minimal k-feasible networks are presented: A greedy variable partitioning based decomposition algorithm and an area-driven variable partitioning decomposition algorithm.

Performance directed delay optimization. A result of area-driven decompositions can be increased circuit delay. Delay re-synthesis techniques are applied on delay critical paths to reduce circuit delay. Critical paths are collapsed into nodes which are then simplified and re-decomposed.

3. PARTIAL COLLAPSE

To reduce logic minimization complexity, a large circuit can be clustered into a smaller set of blocks. Each block consists of a number of collapsed functions, a supernode. Clustering allows individual clusters to be easily decomposed. In our approach, partitioning is used to reduce the number of network nodes.

Maximum fanout free cone (MFFC) based partitioning [Chen et al. 1992] [Cong et al. 1994] is a natural way of clustering nodes in a combinational Boolean network.

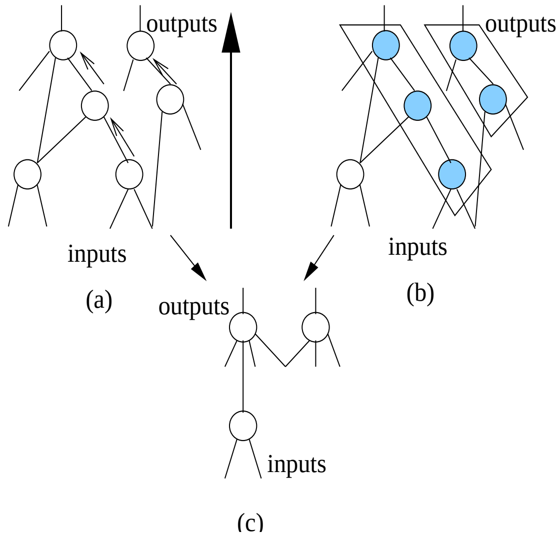

Fig. 3. (a) The new iterative collapse routine, (b) non-iterative MFFC-based collapse, and (c) the final network.

Figure 3(b) illustrates the construction of a MFFC, with the collapsed network shown in Fig. 3(c). For our system, cone input count is used as a collapsing constraint. The number of BDD nodes representing the collapsed node can be used to limit its complexity. Collapsing is performed iteratively until the process converges.

3.1 MFFC based Iterative Eliminate Procedure

ALGORITHM 1. *MFFC-based eliminate*

```
require network (logic network)
begin
  topologically order network nodes from PIs to POs;
  build BDDs for each node; /*partitioned ROBDD for the network*/
  identify MFFCs and identify eliminatable nodes
  while number of collapsible nodes ≠ 0
    while traversed until end of linked list ≠ TRUE
      if (node == collapsible)
        collapse node into its immediate fanout;
    end while
    update network and re-identify all eliminatable nodes
  end while
end
```

Algorithm 1 lists the main operations in the identification and subsequent collapse of nodes. The network is first topologically ordered and traversed from primary inputs to primary outputs. A partitioned ROBDD is then constructed for each node in the network. MFFCs are identified and network nodes that can be collapsed into immediate fanouts are marked and collapsed. These steps are performed iteratively

Table 3. Number of local BDDs to be decomposed for BDS and BDS-pga

	BDS	BDS-pga	
Circuit	#BDDs	#BDDs	% Reduction
C1355	98	78	20.4
C1908	108	58	46.3
C3540	343	200	41.7
C432	65	41	37.0
C499	60	46	23.3
C5315	393	326	17.0
C6288	727	524	28.0
C7552	506	487	4.0
C880	136	80	41.2
dalu	254	223	12.2
des	539	294	45.5
mult32	2494	1066	57.3
pair	446	274	38.6
Total	6169	3697	40.0

until no additional collapsible nodes are found in the network. No gate duplication is performed.

Node collapsing is performed at a Boolean level. Since a BDD is built for each node in the network, node collapsing is reduced to a variable substitute operation (the *bdd_compose* operator is used for this purpose). This collapsing operation can be controlled by user defined limits on BDD size (number of nodes in the BDD) and by the number of cluster inputs.

The use of an iterative MFFC-based iterative eliminate procedure further enhances BDS-pga's optimization capability for large circuits. Table 3 compares results of the new MFFC based eliminate algorithm with those of the original non-iterative implementation in BDS [Yang 2000]. A average reduction of **40%** in the number of network nodes (= no. of BDDs) to be decomposed has been achieved. As the number of BDDs is reduced, the size of each BDD increases, although not significantly. Good decompositions are induced by searching through a larger implementation space. This space corresponds to more BDD variables, but fewer overall BDDs. There is no significant execution time increase for the new eliminate algorithm.

4. GREEDY AND HEURISTIC VARIABLE PARTITIONING BASED DECOMPOSITION

Following MFFC-based eliminate, supernodes are decomposed into a minimum number of k-feasible nodes. First, a greedy variable partitioning technique is used. Subsequently, a variable swapping-based decomposition technique, based on an area cost function, is applied.

Decomposition is the process of breaking a function into subfunctions with smaller fanin. A function, f(X), which depends on n variables, X $= x_1, x_2, \ldots x_n$, can be represented as a new composition function $g(g_1(X_b), \ldots g_m(X_b), X_f)$. X_b is the bound set (BS) and X_f is the free set (FS). Variable partitioning involves computation of the bound and free sets. This partitioning is equivalent to performing a cut on the BDD, using the variables *above* the cut as the bound set and the variables *below* the cut as the free set. The number of edges intersecting the cut line represents the

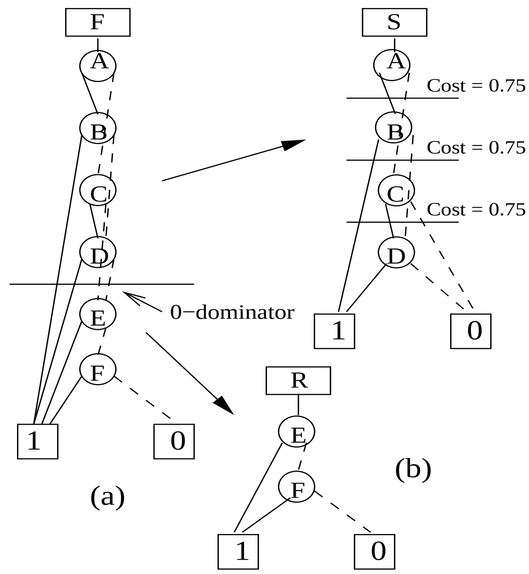

Fig. 4. An example of decomposition with BDS: (a) Algebraic decomposition (disjunctive), (b) subtrahend S and remainder R after the first cut.

number of equivalence classes in Roth-Karp decomposition.

Previously, [Ashenhurst 1959] detailed a procedure for finding a set of variables which cause a simple disjunctive decomposition of a function. [Roth and Karp 1962] proposed a memory efficient algorithm to perform Ashenhurst decompositions and [Lai et al. 1993] described a faster Roth-Karp implementation, using the EVBDD. Stanion and Sechen's method [Stanion and Sechen 1995] implicitly enumerates all BDD cut sets to determine the bound and free sets. They demonstrate that any cut in a BDD can induce a certain function decomposition. Jiang [Jiang et al. 1997] proposed solving the variable partitioning problem with BDDs by selecting lambda set variables in the Roth-Karp decomposition for better LUT utilization. SIS [Sentovich et al. 1992] employs a SOP-based functional decomposition method which greedily selects a non-trivial decomposition of bound set size k. In contrast, Eckl [Legl et al. 1996] proposed a variable partitioning heuristic which selects a **good** bound set size from a number of bound set sizes.

An efficient variable partitioning heuristic must choose a bound and free set such that the smallest number of equivalent classes and subfunctions result from the decomposition. In BDD terms, this approach is equivalent to enumerating all of the cuts possible on the DAG. There are 2^n-1 possible cuts for a function with S_f = n. For computational efficiency, only a subset of cuts, such as cuts with $\leq k$ variables in the bound set, are considered. We describe greedy and area-minimal decomposition in the following three subsections.

4.1 Functional Decomposition using BDS

Although the following analysis is for 3-input LUTs (k = 3), any value of k could be used. The ROBDD of the function $f(a, b, c, d, e, f) = ab + cd + ad + be + e + f$

is shown in Fig. 4(a). BDS decomposes BDDs by searching for the most efficient decomposition. First, it attempts to perform the algebraic decompositions closest to the center. In the absence of algebraic decompositions, Boolean decompositions are performed at the level which yields the least cost, according to the following cost function [Yang 2000],

$$cost = \alpha N + (1 - \alpha)V \tag{1}$$

where N is the ratio of the sum of BDD nodes in the decomposed functions to the number of BDD nodes in the original function; V is the ratio of the number of shared variables to the number of variables in the original function and $\alpha = 0.5$.

The BDS decomposition engine scans the BDD and finds a *0-dominator* located at node E, resulting in an algebraic disjunctive decomposition. The resulting subfunctions are $R = e + f$ and $S = ab + cd + ad$, where $F = S + R$. BDS then decomposes $S = ab + cd + ad$. In Fig. 4(b), the decomposition cost (Eqn. 1) for each level on S has been marked. All decompositions invoke equal cost such that resulting subfunctions from each decomposition are either $\mathcal{S}_{\mathcal{R}} = 2$ and $\mathcal{S}_{\mathcal{S}} = 3$, or vice-versa. BDS decomposes S at the center, below B. The resulting circuit, when mapped using FlowMap, consists of four 3-LUTs and a depth of 2.

4.2 Greedy Variable Partitioning Algorithm

The greedy decomposition engine decomposes the BDD for each internal node into two subfunctions. k-infeasible subfunctions are recursively decomposed until k-feasibility is achieved. The greedy algorithm marks the levels at which decomposition can be performed by visiting each BDD node in a depth-first manner. The algorithm allows decompositions at only those levels which are multiples of k. If $k = 5$, decompositions can be performed at levels $\{0, 5, \ldots, \text{etc.}\}$. The BDD support size (number of variables) determines the level at which the *greedy* decomposition is performed. For a BDD with support size $\leq k$, no decomposition is performed and the BDD is preserved. When BDD support size $\leq 2k$, $2k + 1$, the BDD is decomposed with bound set size $= k$. For a BDD with support size $\leq 3k$, $3k + 1$, the BDD can be decomposed with bound set size $= \frac{no\_levels}{2}$ (via a cut at the center) or $n \times k$, where n is an integer of 1 or greater. For BDDs with support $\geq 3k + 2$, decomposition is performed at the center (i.e. $\frac{no\_levels}{2}$).

The next decomposition step involves the decomposition of a function into two subfunctions. Boolean decompositions result in subfunctions with shared variables (non-disjoint decomposition). The sum of the fanin of each decomposed subfunction may greatly exceed the original fanin, leading to an inefficient decomposition. The cost function,

$$cost = S_{\mathcal{G}} + S_{\mathcal{H}} \leq S_{\mathcal{F}} + S_{\mathcal{F}} \times B \tag{2}$$

where $\mathcal{F}$ is the original function, $\mathcal{G}$ and $\mathcal{H}$ are subfunctions and $B = 0.5$, limits such inefficient decompositions. If a decomposition fails this test, the function is decomposed by cofactoring with respect to the top variable. Experimentation has shown that Boolean decompositions which result in large subfunction fanin lead to a larger number of subsequent decompositions and larger overall area.

Decomposition example: An example of greedy decomposition for $k = 3$ is shown in Figure 5. In Fig 5(a), a cut is greedily performed at level 3. The

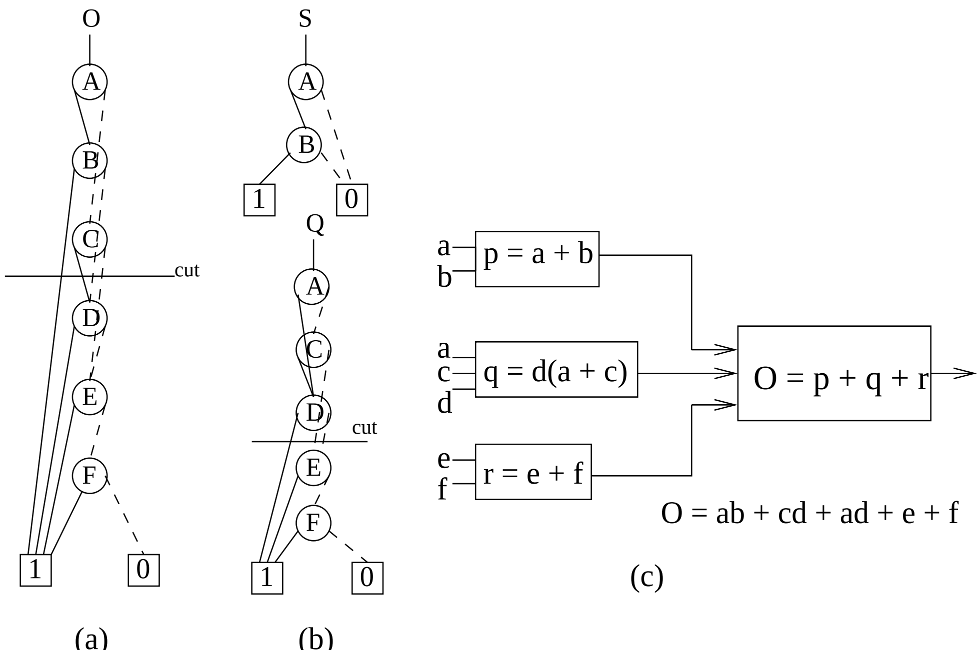

Fig. 5. Greedy decomposition with $k = 3$: (a) the initial *cut* for $S_O = 6$, (b) subtrahend S and remainder Q after the first cut, and (c) the final decomposition.

bound set size is 3, ensuring that at least one subfunction of the decomposition is 3-feasible. The decomposition shown in Fig. 5(b) is Boolean, resulting in a subtrahend $S = ab$ ($\mathcal{S}_{\mathcal{S}}$=$\|(a,b)\| = 2$), and a remainder $Q = cd + ad + e + f$ ($\mathcal{S}_{\mathcal{Q}}$ =$\|(a,c,d,e,f)\| = 5$). The 3-infeasible function (Q) is further decomposed algebraically with the *0-dominator* at node e, and the final decomposition is shown in Fig. 5(c). The result is a network with four 3-LUTs and a topological depth of 2.

4.3 Heuristic Variable Partitioning for Area

The goal of heuristic variable partitioning decomposition for a k-infeasible function is to create a minimum number of k-feasible subfunctions by re-ordering variables around a fixed partition or cut k (i.e. a fixed bound set). This result is achieved by selectively swapping a pair of variables, one variable each from the bound set and the free set. After each swap, an area cost function (ACF) is evaluated to determine the number of subfunctions that would be required for a decomposition at k, with the current variable order. The variable ordering which results in the smallest ACF is chosen.

An FS variable directly connected to a cut edge is a cut-node. In Fig. 6, the BS variables are A and B and the FS variables are C and D. Variables C and *0* are *cut-nodes*. The cardinality of the *cut-node-set*, n, is equal to the number of distinct columns in the Ashenhurst/Roth-Karp decomposition chart and the number of encoding bits (variables) required. Thus, the number of subfunctions

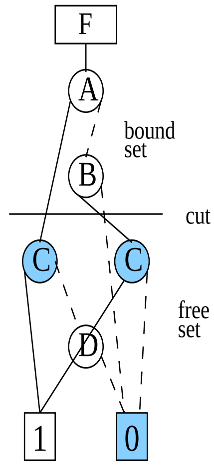

Fig. 6. Illustration of a cut, cut-nodes, and free and bound sets.

resulting from a decomposition can be determined by identifying the *cut-node-set* of the BDD. The number of variables (factored subfunctions) required to encode a given function is $\lceil log_2(n) \rceil$. For the circuit in Fig. 6, the ACF is 1.

Each iteration of the variable partitioning heuristic involves two steps. First, the BDD is scanned using a depth-first search to identify those variables which will most likely lead to a swap reduction in n. The second step involves the calculation of the ACF after the swap. Swapped variables are locked and cannot be swapped until all other variables have been swapped in succeeding iterations. This process continues until all variables are locked or until n decreases. If n does not decrease before all variables are locked, the swapping process is performed for an additional set of iterations. A maximum of $\| BS \| + \| FS \|$ swaps are required to get a good decomposition. After the swaps have taken place, the variable order that has the least ACF is selected.

Identifying the best swap variables: The choice of the FS and BS variables to be swapped is determined by the potential swaps effect on the cardinality of the *cut-node-set*. Intuitively, a FS variable which lies on a large number of paths from the root to the terminals has a large number of incoming edges. This variable generally appears in a large number of function minterms. A swap of this FS variable with any BS variable would generally lead to a reduction in the size of the cut-set. In general, the dependence of the function on the FS variable is not significant.

Definition 1. *A* **FS swap variable** *is the unswapped variable with the most incident edges which resides closest to the cut.*

This result corresponds to a lower column multiplicity in the Ashenhurst/Roth-Karp decomposition chart and leads to fewer encoding variables in the decomposition.

Definition 2. *A BS swap variable is the unswapped variable with the fewest incident edges which resides closest to the cut.*

Table 4. Minterm variable counts measured before the first swap

Variable	Minterms	Incident edges
a	14	–
b	14	2
c	**10**	**2**
d	9	3
e	**9**	**4**
f	5	2

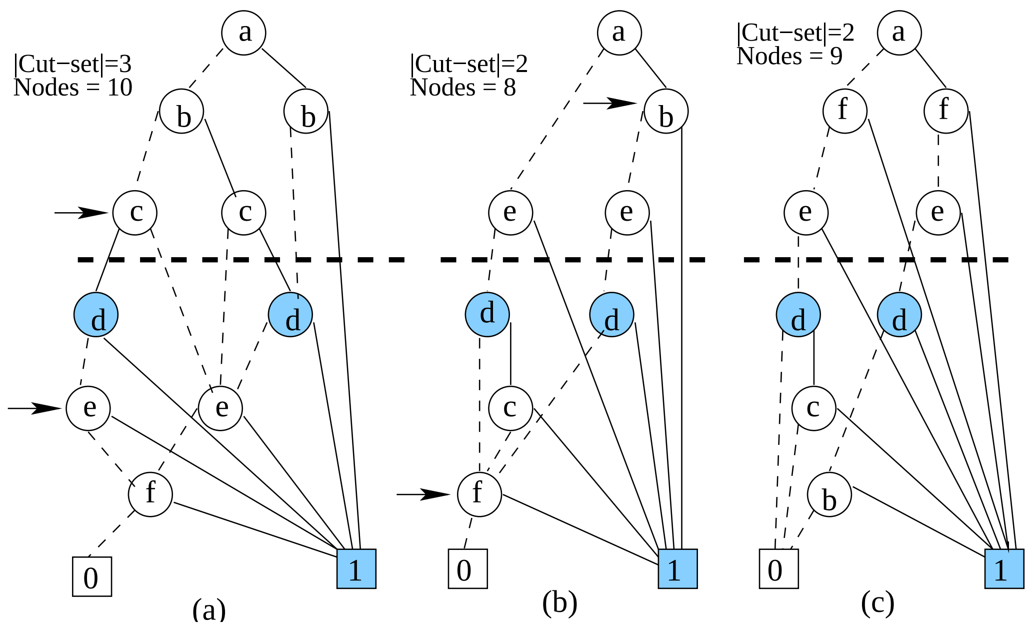

Fig. 7. An illustration of variable swapping. Arrows indicate the swapped variables.

The function $log_2(n)$ is used in Roth-Karp decomposition to determine the number of encoding variables. This measure is highly suitable for a BDD-based decomposition environment. The area cost function is defined as:

$$ACF = \lceil log_2(n) \rceil \tag{3}$$

where n is the cardinality of the *cut-set*. This cost function is an indication of the number of subfunctions that may result from decomposition.

In our heuristic, a cut is placed after the third BDD variable (level) (Fig. 7(a)), so that BS = $\{a,\ b,\ c\}$ and FS = $\{d,\ e,\ f\}$. The BDD in Fig. 7(a) represents the same function as the BDD in Fig. 4. The function has not been reduced, to reflect the swapping mechanism of the decomposition engine. The shaded nodes are the cut_set nodes, and in Fig. 7(a), $\|cut\_set\| = 3$. Thus, ACF = $\lceil log_2(3) \rceil = 2$, which indicates that a minimum of two subfunctions would result from this decomposition. In the next step, the best BS and FS variables for swapping are determined. The

Table 5. Minterm variable counts measured after the first swap

Variable	Minterms	Incident edges
a	8	–
b	**4**	**1**
e	6	2
d	2	2
c	1	1
f	**2**	**3**

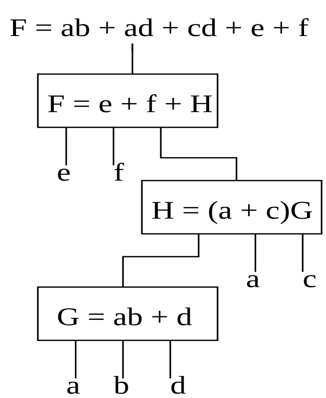

Fig. 8. The final result of decomposition using swapping for the example in Fig. 7.

FS variable with the largest number of incident edges and the BS variable with the smallest number of incident edges are swapped. In the event that more than one variable has the same number of incident edges, the BS and FS variables closest to the cut are swapped. From Table 4, it can be seen that node c is the BS node that appears in the fewest number of minterms (10) of the function in Fig. 7(a) and has the fewest number of incident edges. Additionally, node e is the FS node that appears in the most minterms (9) and has the most incident edges. As shown in Fig. 7(a), node e in the FS and node c in the BS are chosen for swapping.

Algorithm 2. *Heuristic variable partitioning (decomposition) for area*

require *logic network, feasibility factor k*
produce *k-feasible optimized, decomposed network*
begin
  MFFC-based iterative eliminate
  for *each BDD in the network* **do**
    place a cut across a BDD such that bound set size = k or (no. of levels)/2
    repeat
      compute n = cardinality of cut-node-set;
      Select FS swap var as one with most incident edges
      Select BS swap var as one with least incident edges
      Swap the variables and lock them
      compute $ACF = \lceil log_2(n) \rceil$
    until *(all vars swapped or n improves)*
    if *n does not decrease* **then**
      repeat the above process at a different cut across the BDD
    end if
  end for
end

After swapping, a BDD with an ACF of 1 is created. As illustrated in Fig. 7(b), nodes e and c are now locked and cannot be swapped. Nodes f and b are swapped in the next iteration. As shown in Table 5, BS node b appears in the fewest minterms and FS node f appears in the most minterms and has the most incident edges. The new BDD, with an ACF of 1, is shown in Fig. 7(c). The first two variable-pair swaps identify a subfunction, $e + f$. The BDD in Fig. 7(c) is then selected for decomposition, resulting in: $f(a, b, c, d, e, f) = D + H$, where $D = f + f^{'}e$ is the Boolean divisor and $H = ab + ad + cd$ is the quotient. D can be implemented in a 3-LUT, while H requires further decomposition. After iterative decomposition with our approach, the H implementation shown in Fig. 8 is created.

Using the area-driven variable partitioning heuristic, it was possible to map the function onto three 3-LUTs with an overall depth of 3. The ACF was reduced from 2 to 1 in 2 iterations. Using the SIS decomposition command *xl_k_decomp -n 3*, this function is mapped onto 7 LUTs with an overall depth of 3. The algorithm is formally presented in Algorithm 2.

4.4 Comparative Analysis of BDS, Greedy and Area-Minimal Decomposition

Three decomposition heuristics were applied to the function in Fig. 4(a): BDS, greedy decomposition, and area-minimal variable partitioning. Each decomposition approach was followed by technology mapping with FlowMap. Results for each approach were four 3-LUTs (BDS), four 3-LUTs (greedy), and three 3-LUTs (area-minimal). The evaluation of Eqn. 1 for the decompositions performed on Fig. 4(a), Fig. 5(a) and Fig. 7(c) yields decomposition costs: $cost_{BDS} = 0.5$, $cost_{greedy} = 0.67$ and $cost_{area-minimal} = 0.33$. To perform a thorough comparison of the greedy

Table 6. A comparison of greedy and area-minimal heuristics

Ckt.	Greedy		Area-minimal	
	LUTs	delay	LUTs	delay
5xp1	16	2	14	2
9sym	7	3	7	3
9symml	7	3	7	3
C499	154	6	64	4
C880	114	9	108	8
alu2	50	4	41	4
apex6	217	6	186	4
apex7	86	4	71	3
b9	47	5	40	3
count	34	4	26	5
misex1	14	2	14	2
rot	263	9	218	9
C1908	204	12	119	7
C5315	460	11	447	7
Total	**1673**	80	**1362**	64

and area-minimal algorithms, 14 MCNC benchmarks were first optimized using the greedy and area-minimal algorithms and then mapped onto 5-LUT architectures using FlowMap. The results of this experiment are shown in Table 6.

From these results, we conclude:

(1) The BDS decomposition scheme does not account for the eventual implementation of decomposed functions on k-LUTs. In contrast, our greedy implementation restricts decompositions to a bound set size of k. While the greedy approach may lead to inefficient Boolean decompositions, the operation is very fast, due to a small decomposition search space. The greedy method generated more area-efficient circuits than BDS. The bound set size is restricted to k for the area-driven variable partitioning heuristic. A novel variable swapping method coupled with an efficient cost function (ACF) generates BDDs which lead to efficient k-LUT implementations.

(2) The BDS implementation cost (Eqn. 1) is measured on an optimized BDD [Yang 2000]. Although BDD size increases during area-minimal decomposition (as shown in Figs. 7(b) and 7(c)), the implementation cost (using Eqn.1) is reduced from 0.52 to 0.33. This result indicates that a non-reduced BDD (initial representation or final candidate) with more *nodes* may **not necessarily** result in a poorer decomposition, especially for FPGAs.

(3) The evaluation of the BDS cost function is carried out on optimized BDDs, achieved by time-intensive variable reordering. This time-consuming evaluation involves decompositions performed at each level, since results depend on the relative sizes of the generated subfunctions. In constrast, the ACF determines eventual decomposition cost and the dependence of variables in the BS and FS by a simple BDD traversal. This traversal has a time complexity of $O(n)$,

where n is the number of levels in the BDD.

(4) The results in Table 6 indicate that synthesis with the area-minimal heuristic consistently results in circuits that are, on average, 19% smaller and 20% faster than the greedy approach. This is a result of the intelligent variable swapping technique present in the area-minimal approach. All further results are based on the area-minimal heuristic decomposition technique.

5. DELAY RE-SYNTHESIS

After initial optimization, re-synthesis techniques can be applied to circuits to improve circuit delay. Speed-up [Singh et al. 1988] was one of the first delay reduction approaches for combinational networks which combined sub-network collapsing with subsequent decomposition. A number of similar performance directed re-synthesis techniques for FPGAs [Murgai et al. 1995][Lai et al. 1993][Legl et al. 1996] [Cong and Ding 1993] have also been studied.

5.1 The Delay Algorithm

The input to our delay-based algorithm is a circuit that has been optimized for area in the area-minimization pass of BDS-pga. Starting at the primary outputs, the algorithm identifies critical paths, depending on the depth of the transitive fanin.[1] Collapsing is limited to the critical path since indiscriminate collapsing can potentially increase the fanin of the resulting nodes and increase the final implementation cost after technology mapping. Although node collapsing reduces the depth of the primary output node, the collapsed node may become k-infeasible. To generate k-feasible nodes, the collapsed node is simplified with Espresso [Brayton et al. 1984], a heuristic two-level logic minimizer. The simplified node is then re-decomposed, based on our variable partitioning decomposition algorithm.

5.1.1 *Determining Critical Paths.* After the area-optimized network is parsed and depths are assigned, $\mathcal{L}$ is determined. Starting at the primary output(s) with the largest depth(s), a post-order traversal to the primary inputs is performed. Figure 9(a) illustrates the critical path from primary output *PO1.* Nodes are assigned character names; integers represent the depths at inputs/outputs. To determine $\mathcal{L}_{\Upsilon}$, all primary outputs with largest depth are considered.

During $\mathcal{L}_{\Upsilon}$ evaluation, a node n that has been previously marked as critical may be encountered. Instead of traversing the ϵ-critical paths from this node, the critical path $\mathcal{L}$ is followed, avoiding the *sub-critical* paths in proximity to the critical one in question. For example, Fig. 9(c) depicts the critical path marked in a *previous* iteration. The arrow at the left in Fig. 9(c) indicates the sub-critical path that must be traversed and marked. This issue is resolved by either traversing the next most critical path $L_{\epsilon-eps}$ from node n, or by exiting from the routine at this point.

5.1.2 *Level Reduction by Partial Collapse.* After the determination of critical paths, nodes are partially collapsed into their critical fanouts. A partially collapsed circuit is shown in Fig. 9(b). Since nodes are collapsed into others on the critical

[1] Although false path identification is not currently included in **BDS-pga**, it could be easily incorporated into our synthesis framework.

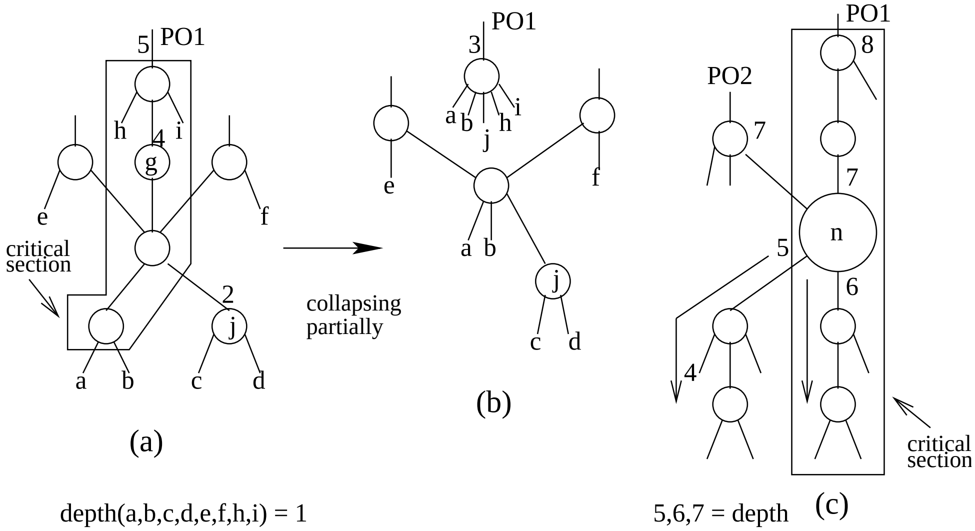

Fig. 9. Examples of (a) critical path determination, (b) collapsing, and (c) sub-critical path determination.

path, the duplication of multiple-fanout nodes is required. In Fig. 9(a), the depth of node **PO1** is 5. In Fig. 9(b), after partial collapse, the largest depth at the input of PO1 (j) is 2. The depth at the output of node **PO1** is 3.

5.1.3 *Simplification of collapsed nodes using Espresso.* After the collapse phase, supernodes are simplified using the two-level minimizer, Espresso. Logic simplification aims to reduce the fanin (support size) of supernodes to induce a delay-efficient decomposition. BDS does not have an efficient logic minimization capability using don't cares. Although the *BDD_restrict* [Bryant 1986] algorithm has been proposed for this purpose [Yang et al. 1999], the results have not been satisfactory. Due to the relatively small size of the supernodes, two-level logic minimization techniques, like Espresso, can be used to generate simplified supernodes with fewer minterms. The resulting k-feasible node is preserved, resulting in a decrease in critical path depth.

5.1.4 *Re-decomposition.* After simplification, the simplified k-infeasible nodes are re-decomposed using heuristic variable partitioning. A decomposed sub-network of k-feasible subfunctions with minimum delay is created.

6. RESULTS

Our **BDS-pga** system has been implemented and tested using benchmark circuits from the ISCAS 89 [ISCAS 1989] and MCNC [MCNC 1991] benchmark suites. To illustrate the benefits of our approach, BDS-pga area and delay results are compared to results generated by SIS and BoolMap [Legl et al. 1996], two academic synthesis packages, and *Quartus, version v2000.02*, a commercial synthesis tool.

Table 7. Area results for BoolMap-Area, SIS optimization and BDS-pga decomposition.

Ckt.	BoolMap-Area			scr.rug+FlowMap			BDS-pga+FlowMap		
	LUTs	delay	CPU (s)	LUTs	delay	CPU (s)	LUTs	delay	CPU (s)
5xp1	13	2	0.6	40	8	8	14	3	1.1
9sym	8	3	0.4	108	6	44	7	3	1.2
9symm	8	3	0.3	9	6	33	7	3	1.3
C499	98	5	40.4	66	4	50	70	5	1.2
C880	121	11	5.4	136	9	18	103	8	2.0
alu2	46	5	126.0	134	13	164	41	4	4.1
alu4	150	11	101.0	235	11	1614	190	7	25.2
apex6	152	6	12.5	214	9	30	186	4	7.8
apex7	61	5	8.0	70	5	8	71	3	3.3
b9	43	3	0.4	53	4	9	40	3	2.3
clip	15	2	0.6	90	7	43	30	4	12.5
count	31	7	1.6	31	5	4	26	5	1.9
des	1462	9	86.8	1396	8	595	909	4	55.7
duke2	187	8	199.0	169	6	36	173	8	6.9
misex	13	2	0.2	17	4	2	14	2	7.1
rd84	10	2	2.0	146	6	71	13	3	3.3
rot	347	19	86.2	250	11	46	223	10	10.6
vg2	31	4	9.3	40	5	17	12	3	5.0
z4ml	5	2	0.2	5	2	3	5	2	1.4
t481	5	3	30.1	177	8	116	5	2	2.6
C1355	80	6	4.9	66	4	53	64	4	5.2
C1908	130	12	40.5	115	10	64	123	7	7.3
C5315	545	13	37.2	522	9	100	435	7	23
Total	3561	143	1098.4	3651	160	3065.0	**2761**	**95**	191.8
Norm	0.98	0.89	0.36	1	1	1	**0.75**	**0.59**	**0.062**

6.1 Experimental Results: Area Optimization

Synthesis experiments using SIS, BoolMap, and BDS-pga were conducted using multi-level optimization followed by technology mapping. The RASP script [Cong et al. 1996], which includes FlowMap [Cong and Ding 1994] and FlowSYN [Cong and Ding 1993], was used for technology mapping in all cases. LUT input size k was set to 5 for these experiments. All experiments were run on a 633 MHz Celeron-based PC with 128 MB of memory.

For experiments using the TOS synthesis system (BoolMap) [Legl et al. 1996], area was measured in terms of the number of LUTs, and the delay was measured in terms of LUT critical path depth. MCNC benchmarks were first collapsed and then decomposed to k-feasible networks with the optimization script *mmap_h_a_5.scr* [Eckl et al. 1996] [Legl et al. 1996]. ISCAS benchmarks, which could not be fully collapsed, were initially decomposed using the *smap_h_a_5.scr* script, followed by the *reduce_depth* procedure. The multi-output decomposition and mapping script *mmap_h_a_5.scr* was then used to generate an optimized, mapped network. Results from these experiments are shown under the **BoolMap-Area** column in Table 7.

For SIS, the benchmark circuits were first optimized using the script *script.rugged*. The resulting optimized netlists were subsequently mapped onto FPGAs using

Table 8. Delay re-synthesis results for BoolMap-Delay, SIS optimization and BDS-pga re-decomposition

Ckt.	BoolMap-d			scr.rug + scr.delay+FlowMap			BDS-pga+ FlowMap	
	LUTs	delay	CPU(s)	LUTs	delay	CPU(s)	LUTs	delay
5xp1	13	2	0.9	31	4	9	15	2
9sym	7	3	0.4	74	6	21	7	3
9symml	7	3	0.3	89	6	27	7	3
C499	102	4	107.1	92	4	40	64	4
C880	134	8	13.3	174	10	105	108	8
alu2	50	5	62.7	168	10	91	41	4
apex6	188	4	12.5	210	6	41	186	4
apex7	78	3	8.0	68	4	10	71	3
b9	41	3	0.4	43	3	4	40	3
clip	15	2	0.6	70	7	22	30	4
count	42	2	1.6	50	4	7	26	5
duke2	192	5	199.0	200	6	37	169	7
misex1	15	2	0.2	6	2	2.2	14	2
rd84	10	2	2.0	99	5	36	13	3
rot	244	6	-	300	6	97	218	9
vg2	30	4	9.3	41	4	10	12	3
z4ml	5	2	0.2	6	2	2	5	2
t481	5	3	30.1	160	7	616	5	2
C1355	98	5	10.7	4	114	17	65	4
C1908	137	7	97.0	186	9	99	119	7
C5315	672	9	154.0	711	10	184	447	7
C7552	729	9	102.6	602	11	114	631	12
Total	**2814**	103	813	3482	130	1591	**2293**	93
Norm	**0.81**	0.79	0.51	1	1	1	**0.66**	0.72

FlowMap. SIS results are shown under the **scr.rug+FlowMap** column in Table 7.

For *BDS-pga*, circuits were partially collapsed to produce clusters of supernodes using the MFFC-based approach described in Section 3. Logic decomposition was performed using the **heuristic variable partitioning technique** to produce a k-feasible network. Following logic optimization, technology mapping was performed using FlowMap. **BDS-pga** results are shown under the **BDS-pga+FlowMap** column in Table 7.

The RASP script (FlowMap) has been modified to preserve the nature of the decompositions obtained by **BDS-pga**. RASP decomposes network nodes into 2-feasible AND/OR gates as a pre-processing step. This step can potentially undo the optimization characteristics of **BDS-pga**. Additionally, **BDS-pga** identifies AND/XNOR decompositions. These decompositions could be undone by RASP, since it decomposes all complex gates into a network of AND-OR gates. To preserve AND/XNOR decompositions, the RASP technology decomposition routine

Table 9. Unsynthesizable circuits for BoolMap and SIS

Ckt.	BoolMap-Area		scr.rug + FlowMap		BDS-pga+FlowMap	
	LUT	delay	LUTs	delay	LUTs	delay
Area Oriented Synthesis						
C3540	-	-	678	17	311	15
C7552	696	13	671	13	631	12
Delay Oriented Synthesis						
alu4	264	7	-	-	190	7
des	594	3	1582	7	909	4
C3540	-	-	542	12	324	13
pdc	-	-	4605	11	4012	11

was disabled.[2]

Table 7 shows results for 23 MCNC and ISCAS benchmarks. Comparison of LUT area results for BDS-pga in Table 7 to those for BDS in Table 2 indicates a 11.8% area improvement for BDS-pga over BDS. The results demonstrate that the optimization provided by BDS-pga is significant. Overall, **BDS-pga** generates circuits with fewer LUTs than SIS and BoolMap. For almost all benchmarks, circuit delay (topological depth in LUTs) is smallest when **BDS-pga** is used. For most benchmarks, **BDS-pga** CPU times are an order of magnitude smaller than for either SIS and BoolMap. SIS was unable to optimize C7552, while BoolMap failed to synthesize C3450 in acceptable time (almost 5 hrs). As shown in Table 9, **BDS-pga** was able to efficiently synthesize both circuits in a matter of seconds.

6.2 Experimental Results: Delay Optimization

After area-based comparison, **BDS-pga** was compared to SIS and BoolMap using delay-synthesis techniques for all tool suites. For SIS, script *script.delay* was applied after *script.rugged* to generate delay-optimal circuits. FlowMap was then used to complete technology mapping. For BoolMap, delay-optimization scripts [Eckl et al. 1996] were used to generate delay-optimized circuits. For **BDS-pga**, resynthesis along delay critical paths was performed for the area-optimized circuits from the previous experiment. As described in Section 5, this approach partially collapses the critical path. FlowMap completes technology mapping. Results for all three approaches are shown in Table 8.

It can be seen from the table that **BDS-pga** outperforms SIS for delay-optimization and compares favourably with BoolMap. As shown in Table 9, under the **delay** heading, several benchmarks could not be synthesized for delay by BoolMap and SIS, but were handled successfully by **BDS-pga**. A comparison of delay results in Table 7 and Table 8 shows that the delay resynthesis step in **BDS-pga** almost always reduces delay. In no case did delay resynthesis increase the delay of the network. Also, it can be noted that generally delay improvement is not achieved at

[2] In experiments with **scr.rug+FlowMap**, the 2-input AND/OR decomposition feature of FlowMap was not disabled.

Table 10. Area results for Quartus and BDS-pga for the Apex architecture ($k = 4$).

Ckt.	Quartus			BDS-pga+FlowMap		
	LUTs	delay	CPU time (s)	LUTs	delay	CPU time (s)
des	1501	9	436	1592	5	50
C1355	98	6	44	94	4	9
C1908	223	8	69	162	8	3
C5315	640	14	271	582	8	14
C3540	426	18	154	556	11	28
C7552	788	15	372	769	11	19

Table 11. Delay results for Quartus and BDS-pga for the Apex architecture ($k = 4$)

Ckt.	Quartus			BDS-pga+FlowMap		
	LUTs	delay	CPU time (s)	LUTs	delay	CPU time (s)
des	2055	8	502	1590	5	53
C1355	86	5	45	90	4	4
C1908	294	7	73	163	7	3
C5315	833	12	253	588	7	14
C3540	575	18	198	556	11	28
C7522	1013	14	394	770	10	19

the expense of area. The circuits synthesized by our approach deliver better area and performance than previous approaches.

6.3 Comparative Results for BDS-pga and Quartus

Comparative area minimization results for FPGA and Quartus are presented in Table 10. For these experiments, the value of k, the LUT input count, was kept to 4, the LUT size in Altera's Apex series of devices. Similar to previous experiments, FlowMap's two-input AND/OR decomposition was disabled to preserve BDS-pga's k-feasible decomposition. From the table, it can be noted that for benchmarks *C1355, C1908, C5315* and *C7522*, BDS-pga produces better area optimized results than Quartus. Moreover, the delay of all circuits synthesized using BDS-pga and FlowMap is less than delay for circuits synthesized by Quartus and the CPU times for BDS-pga and FlowMap combined are orders of magnitude smaller than those required by Quartus[3]. Delay resynthesis results shown in Table 11 indicate that the combination of BDS-pga and FlowMap produces improved or equal delay for all circuits versus Quartus. Additionally, LUT area is reduced by **BDS-pga** for all circuits, except one.

[3]Time utilized by Quartus consists of the time required to build the circuit database and perform logic synthesis and mapping. For our experiments, only the time required for logic synthesis and mapping is reported to provide a fair comparison with BDS-pga.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented **BDS-pga**, a BDD-based, FPGA-specific logic synthesis system. This approach unites multi-level logic transformation techniques with FPGA-specific logic decomposition to form a novel synthesis framework. Decisions are made earlier in the synthesis process to promote mapping to LUT-based FPGAs. The resulting synthesized circuits exhibit superior area and performance characteristics when compared to circuits synthesized by conventional FPGA synthesis tools. By using an efficient BDD-decomposition engine, we are able to decompose large designs quickly, without sacrificing the quality of the resulting implementation.

One of the limitations of our tool is the lack of support for logic simplification with don't care sets using BDDs. While *restrict* and *constrain* operators have been proposed to simplify a specific BDD with respect to another, no satisfactory solution has been found. This limitation has forced us to use Espresso. Although this tool robustly handles relatively large designs, it often requires significant computation time. We are currently investigating several *implicit* logic minimization schemes for incorporation within our synthesis framework. Although our approach attempts to decompose logic into k-feasible networks, technology mapping is ultimately carried out as a post-processing step. Analogous to the approach presented in [Chen and Cong 2001], it would be desirable to analyze a set of k-feasible decompositions during technology mapping to derive a better mapped solution.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Maciej Ciesielski (Univ of Massachusetts, Amherst) and Congguang Yang (Chameleon Systems) for providing their logic synthesis tool, BDS, for use in our work. The authors would also like to thank Klaus Eckl at the Technical Institute of Munich for granting us a license to use BoolMap.

REFERENCES

Ashenhurst, R. L. 1959. The decomposition of switching functions. In *Proc. International Symp. Theory of Switching Functions*, 74–116.

Babba, B. and Crastes, M. 1992. Automatic synthesis on table lookup-based FPGAs. In *Proc. Euro-ASIC* (May), 25–31.

Brayton, R., Hachtel, G., McMullen, C., and Sangiovanni-Vincentelli, A. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer, Boston.

Brayton, R., Rudell, R., Sangiovanni-Vincentelli, A., and Wang, A. 1987. MIS: A multiple-level logic optimization system. *IEEE Trans. Computer-aided Design. 6*, 6 (Nov.), 1062–1081.

Brglez, F., Bryan, D., and Kozminski, K. 1989. Combinational profiles of sequential benchmark circuits. In *Proc. IEEE International Symp. Circuits and Systems* (May), 1929–1934.

Bryant, R. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. C-35*, 8 (Aug.), 677–691.

Chang, S., Marek-Sadowska, M., and Hwang, T. 1996. Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams. *Proc. IEEE Trans. Computer-aided Design. 15*, 10 (Oct.), 1226–1236.

CHEN, G. AND CONG, J. 2001. Simultaneous logic decomposition with technology mapping in FPGA designs. In *Proc. International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb), 48–55.

CHEN, K., CONG, J., DING, Y., KAHNG, A., AND TRAJMAR, P. 1992. DAG-Map: Graph-based FPGA technology mapping for delay optimization. *IEEE Design and Test of Computers.* (Sept.), 7–20.

CONG, J. AND DING, Y. 1993. Beyond the combinatorial limit in depth minimization for LUT-Based FPGA designs. In *Proc. IEEE/ACM International. Conf. on CAD* (Santa Clara, CA, Nov.), 110–114.

CONG, J. AND DING, Y. 1994. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. Computer-Aided Design 13*, 1 (Jan.), 1–12.

CONG, J., LI, Z., AND BAGRODIA, R. 1994. Acyclic multi-way partitioning of boolean networks. In *Proc. IEEE/ACM Design Automation Conference* (June), 670–675.

CONG, J., PECK, J., AND DING, Y. 1996. RASP: A general logic synthesis system for SRAM-based FPGAs. In *Proc. International Symposium on Field Programmable Gate Arrays* (Monterey, Calif., Feb.), 137–143.

COUDERT, O. AND MADRE, J. 1990. A unified framework for the formal verification of sequential circuits. In *Proc. International Conference on Computer Aided Design*, 126–129.

DEMICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Hightstown, NJ

ECKL, K., LEGL, C., AND WURTH, B. 1996. *TOS Version 2.2: User Manual*. Institute of Elec. Design Automation, Tech. Univ. of Munich.

FILO, D., YANG, J., MAILHOT, F., AND DEMICHELI, G. 1991. Technology mapping for two output RAM-based FPGAs. In *Proc. European Design Automation Conf.*, 534–538.

FRANCIS, R., ROSE, J., AND CHUNG, K. 1990. Chortle: A technology mapping algorithm for lookup table-based field programmable gate arrays. In *Proc. Design Automation Conf.* (June), 613–619.

FRANCIS, R., ROSE, J., AND VRANESIC, Z. 1995. Chortle-crf: Fast technology mapping for lookup table-based FPGAs. In *Proc. International Symp. Theory of Switching Functions*, 74–116.

HACTEL, G. D. AND SOMENZI, F. 1996. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, Dordrecht, The Netherlands.

JIANG, J.-H., JOUT, J.-Y., HUANG, J.-D., AND WEI, J.-S. 1997. A variable partitioning algorithm of BDDs for FPGA technology mapping. *IEIEC Trans. Fundamentals of Electronics E80*, 10 (Oct.), 1813–1819.

KARPLUS, K. 1991. XMAP: A technology mapper for table-lookup based FPGAs. In *Proc. Design Automation Conf.* (June), 240–243.

LAI, Y., PEDRAM, M., AND VRUDHULA, S. 1993. BDD based decomposition of logic functions with application to FPGA synthesis. In *Proc. Design Automation Conf.* (June), 642–647.

LEGL, C., WURTH, B., AND ECKL, K. 1996. A boolean approach to performance-directed technology mapping for LUT-based FPGA designs. In *Proc. Design Automation Conf.* (June), 74–116.

MURGAI, R., BRAYTON, R., AND SANGIOVANNI-VENCENTELLI, A. 1995. *Logic Synthesis for Field-Programmable Gate Arrays.* Kluwer, Boston

MURGAI, R., NISHIZAKI, Y., BRAYTON, R., AND SANGIOVANNI-VINCENTELLI, A. 1990. Logic synthesis for programmable gate arrays. In *Proc. Design Automation Conf.* (June), 620–625.

Narayan, A., Jain, J., Fujita, M., and Sangiovanni-Vincentelli, A. 1996. Partitioned-ROBDDs: A compact canonical and efficient representation for boolean functions. In *Proc. Int. Conf. Computer Aided Design* (Nov.), 547–554.

Roth, J. and Karp, R. 1962. Minimization over boolean graphs. *IBM J. of Research and Development 6*, 227–238.

Savage, J. E. 1976. *The Complexity of Computing*. Wiley, New York

Sawada, H., Suyama, T., and Nagoya, A. 1995. Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization. In *Proc. Int. Conf. Computer-Aided Design* (Nov.), 54–59.

Sawkar, P. and Thomas, D. 1992. Area and delay mapping for table-look up based field programmable gate arrays. In *Proc. Design Automation Conf.* (June), 368–373.

Sentovich, E., Singh, K., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P., Brayton, R., and Sangiovanni-Vincentelli, A. 1992. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41 (March), Dept. of EECS, Univ. of Calif., Berkeley.

Singh, K., Wang, A., Brayton, R., and Sangiovanni-Vincentelli, A. 1988. Timing optimization of combinational logic. In *Proc. International Conf. Computer Aided Design* (Nov.), 282–285.

Stanion, T. and Sechen, C. 1995. A method for finding good Ashenhurst decompositions and its application to FPGA synthesis. In *Proc. Design Automation Conf.* (June), 74–116.

Trimberger, S. 1994. *Field Programmable Gate Array Technology*. Kluwer, Boston.

Yang, C. 2000. *BDD-Based Logic Synthesis System*. Ph. D. thesis, Univ. of Massachusetts, Amherst, Department of Electrical and Computer Engineering.

Yang, C., Singhal, V., and Ciesielski, M. 1999. BDD decomposition for efficient logic synthesis. In *Proc. International Conf. Computer Design* (Oct.).

Yang, C., Singhal, V., and Ciesielski, M. 2000. BDS: A BDD-based logic synthesis system. In *Proc. Design Automation Conf.* (June), 92–97.

Yang, S. 1991. Logic Synthesis and Optimization Benchmarks, Version 3.0. Technical Report, Microelectronics Centre of North Carolina.